MSPICE Manual v2.0

Justin Reiher

Contents

1 Introduction **5**
1.1 MSPICE Setup 6
1.2 MSPICE Software Struture 6
1.3 Citing MSPICE 7

2 MSPICE: Quick Start Guide Three Stage Ring Oscillator **7**
2.1 Defining the circuits 9
2.1.1 Inverter `INV4.m` 10
2.1.2 Three stage ring oscillator: `THREE_RING_OSC.m` 15
2.2 Simulating the Three Stage Ring Oscillator 18
2.2.1 Creating voltage sources 19
2.2.2 Creating the `circuit` 19
2.2.3 Creating the `testbench` options 19
2.2.4 Creating the `testbench` 20
2.2.5 Simulating `THREE_RING_OSC.m` 21

3 Synchronizer Metastability Simulation and Analysis Guide: Passgate Latch Synchronizer **22**
3.1 Creating the voltage sources 23
3.2 The `nestedBisection testbench` 24
3.2.1 Defining `stopToDigitalSim.m` 24
3.2.2 Defining `nestedBisectionStop.m` 25
3.2.3 Running the nested bisection algorithm 27
3.3 The `nestedBisectionAnalysis testbench` 27
3.3.1 Running the analysis algorithm 27

4 Optimizing Synchronizer **28**
4.1 `populateADoptimizationVar.m` 29
4.1.1 Two flip-flop passgate synchronizer example 30
4.2 `expandW.m` 33
4.3 Optimization Constraints 34

5 MSPICE Circuit **34**
5.1 Circuit `TEMPLATE.m` 34
5.2 `circuit.m` class 35
5.2.1 `circuit` constructor (protected) 36
5.2.2 `add_port` (protected) 36
5.2.3 `add_element` (protected) 36

5.2.4 connect (protected) . . . 36
5.2.5 finalize (protected) . . . 37
5.2.6 ifc_simu . . . 37
5.2.7 is_vsrc . . . 37
5.2.8 getCircuitMap . . . 37
5.2.9 getDevices . . . 37
5.2.10 getDevicesConnectedToNode . . . 37
5.2.11 getDevicesByIndex . . . 38
5.2.12 getElementWiseParameters (private) . . . 38
5.2.13 vectorize . . . 38
5.2.14 flatten . . . 38
5.2.15 is_leaf . . . 38
5.2.16 elemNum . . . 38
5.2.17 find_port_index . . . 38
5.2.18 find_node_name . . . 39
5.2.19 find_node_name_short . . . 39
5.2.20 find_path . . . 39
5.2.21 print_circuit_tree . . . 39
5.2.22 print_status . . . 39
5.2.23 print_nodes . . . 39
5.2.24 create_path . . . 39
5.2.25 add_path . . . 39
5.3 coho_leaf.m class . . . 40
5.3.1 resistor . . . 40
5.3.2 capacitor . . . 40
5.3.3 inductor . . . 41
5.3.4 pmos . . . 41
5.3.5 nmos . . . 42
5.4 coho_vsrc.m class . . . 42
5.4.1 vsrc . . . 42
5.4.2 vpulse . . . 42
5.4.3 vsrcLinear . . . 43
5.4.4 vsin . . . 43
5.4.5 vtanh . . . 43
5.4.6 vtanhClock . . . 44
5.4.7 vtanhDelay . . . 45

6 MSPICE Testbench **45**
6.1 testbench.m class . . . 45
6.1.1 Testbench options: tbOptions . . . 46
6.1.2 simulate . . . 47
6.1.3 dV_ode . . . 48
6.1.4 packageTestbenchOption . . . 48
6.1.5 getTestbenchOptions . . . 48
6.1.6 setTestbenchOptions . . . 48
6.1.7 setSimTimeVoltage . . . 48
6.1.8 dim . . . 48
6.1.9 name . . . 49
6.1.10 dut . . . 49
6.1.11 inputNum . . . 49

6.1.12 inputs . . . 49
6.1.13 inputNodes . . . 49
6.1.14 find_port_index . . . 49
6.1.15 getIdsDevices . . . 49
6.1.16 explainNodes . . . 50
6.1.17 setOdeSave . . . 50
6.1.18 getOdeSave . . . 50
6.1.19 eval_vs . . . 50
6.1.20 is_src . . . 50
6.1.21 computeMapMatrix . . . 51
6.1.22 vfull (private) . . . 51
6.2 nestedBisection.m class . . . 51
6.2.1 nested bisection options: tbOptions . . . 53
6.2.2 simulate . . . 54
6.2.3 stopCriteria (protected) . . . 55
6.2.4 stopToDigitalSim.m . . . 56
6.2.5 nestedBisectionStop.m . . . 56
6.2.6 dV_ode . . . 57
6.2.7 findClockEdges . . . 57
6.2.8 findNextBracket (protected) . . . 57
6.2.9 findNextStartStatesAndTime (protected) . . . 57
6.2.10 bisectionPlotSetup (protected) . . . 58
6.2.11 bisectionPlotV (protected) . . . 58
6.2.12 bisectionPlotBeta (protected) . . . 59
6.2.13 vfull (protected) . . . 59
6.2.14 setInterval (protected) . . . 59
6.3 nestedBisectionWithKick.m class . . . 59
6.4 nestedBisectionAnalysis.m class . . . 60
6.4.1 nestedBisectionAnalysis . . . 60
6.4.2 nested bisection analysis options: tbOptions . . . 61
6.4.3 dbeta_ode . . . 61
6.4.4 dwdt_ode . . . 62
6.4.5 gainSync . . . 62
6.4.6 splineBisection . . . 62
6.4.7 getVfull . . . 62
6.4.8 computeJacVdot . . . 62
6.4.9 computeVdot . . . 63
6.4.10 computeJac . . . 63
6.4.11 computeMapMatrixDevice . . . 64
6.4.12 computeMapMatrixDeviceTx . . . 64
6.4.13 dGdw . . . 64
6.4.14 populateADoptimizationVar.m . . . 65
6.4.15 expandW.m . . . 65
6.4.16 gamma_ode . . . 65
6.4.17 vfull (private) . . . 65
6.4.18 betaRenorm (private) . . . 66

7 MSPICE Models **66**
7.1 Model Dictionary class 66
7.1.1 `getModel` 67
7.1.2 `getModelConfig` 67
7.2 Model interface 68
7.2.1 `I` 68
7.2.2 `IdevRes` 69
7.2.3 `C` 69
7.2.4 `iPMOS` 70
7.2.5 `iNMOS` 70
7.2.6 `iL` 70
7.2.7 `iC` 70
7.2.8 `iR` 71
7.2.9 `cPMOS` 71
7.2.10 `cNMOS` 71
7.2.11 `cL` 71
7.2.12 `cC` 71
7.2.13 `cR` 71
7.2.14 `iresPMOS` 72
7.2.15 `iresNMOS` 72
7.2.16 `iresL` 72
7.2.17 `iresC` 72
7.2.18 `iresR` 72
7.3 Model cards 72
7.3.1 `junctionDrainCap.m` 73
7.3.2 `interpModel.m` 73
7.3.3 `ekvModel.m` 73
7.3.4 `mvsModel.m` 74
7.3.5 `mvsADModel.m` 74

8 MSPICE Library contents **74**
8.1 Circuits 74
8.2 Components 75
8.3 Circuit leaves 75
8.4 Voltage Sources 75

9 Future Work and Improvements **76**

Preface

MSPICE is based on the work of Yan [21] and has been extended as part of a MSc thesis by Reiher [17]. The original repository can be found at: `https://github.com/coho-tools/ccm`. The extension provides an implementation of the nested bisection algorithm for synchronizer metastability simulations developed by Yang & Greenstreet [23] and its analysis by Reiher *et al.* [15, 16]. Additional debugging, diagnostics and plotting routines have been added to facilitate information extraction. The extensions have been tested and run using MATLAB R2017b [12] and known to be working with MATLAB R2019b.

The transistor models are derived from the Predictive Technology Model (PTM) developed by Arizona State University [8]. In particular the examples and models found in this manual make use of the PTM 45nmHP model card.

This manual is intended to provide insight on how to create circuits, run simulations for general circuits and synchronizers. The options available to the user are outlined in this manual with detailed examples as guides to follow to get the user started on their own experiments.

The synchronizer analysis, small-signal model creation and optimization routines outlined in this manual require the Automatic Differentiation (AD) package found at `http://www.ti3.tu-harburg.de/intlab/` by Rump [18].

1 Introduction

The MSPICE code base is a MATLAB implementation of a circuit simulator such as LTSPICE [1], HSPICE [7] or ngSPICE [19]. MSPICE is open source and extensible to meet your needs. The version outlined in this manual seeks to facilitate further experiments and extensions by providing detailed explanations to the features and routines found within, while also providing insight on how one might go about making their own extensions.

MSPICE is not intended to compete with other open source or commercially available SPICE programs but is designed to be useful in the exploration of new circuit analysis techniques by providing a framework which gives users the ability to use the wealth of routines and functions found within MATLAB. The framework gives the user complete access to all data structures (with the appropriate methods) to be manipulated as seen fit. MSPICE should be viewed as a testing ground to try new algorithms and explore their consequences in an easy to debug framework before spending the effort in improving the computational efficiency in a language such as C/C++ or before extending other SPICE programs.

The manual is organized into quick start guides with instructions on how to use the framework as well as a comprehensive list of methods that explain what they do and how they work.

The intent of this manual is to be used as a reference with some detailed examples to unpack some of the steps necessary in using MSPICE. However I recommend browsing the files running some scripts and refer to the manual to obtain details about specific parts of the code base, or to consult it if there is information you want to obtain from a circuit, simulation or synchronizer metastability simulation that is not obvious. It is quite possible that what you

are looking for has been implemented.

If it has not, please feel free to add your own methods and routines!

1.1 MSPICE Setup

First clone the repository to your computer.

```
git clone https://github.com/justinreiher/ccm.git
```

The above will clone the repository to your computer `./ccm`. All the contents of MSPICE and routines are found within.

1. Open MATLAB, from the command line run:
   `addpath(genpath('<<insert_your_path>>/ccm'))`
   Where `<<insert_your_path>>` is the path to the `ccm` folder which has just been cloned. This will add to the MATLAB path all folder and sub-folders within the folder `ccm`.

2. Alternatively you can use MATLAB's gui and click on the *Set Path* button under the *Home* tab, press *Add with Subfolders...* and navigate to where the `ccm` folder was cloned. Press *Save* and now when you open MATLAB you should have all `ccm` functions and routines available.

WARNING: If your MATLAB path includes the `RF Toolbox`, it will interfere with the MSPICE `circuit.m` class. To prevent this either remove the `RF Toolbox` or remove from MATLAB's path:

```
./MATLAB/201xx/mcr/toolbox/rf/rf
```

and save the updated MATLAB path.

1.2 MSPICE Software Struture

MSPICE is broken into three main software classes:

1. `circuit` classes
2. `testbench` classes
3. `model` classes

Figure 1 illustrates the class organization of MSPICE. The connection between the `testbench` and `model` is bridged through the `modelDictionary` class which contains all available models. The `model` class is actually an interface which must be implemented by any new model that gets added to MSPICE.

The `circuit` class is used to create circuit definitions and voltage sources used at simulation time in the `testbench` class.

A `testbench` takes a `circuit` and `modelDictionary` class which then allows the user to simulate the circuit and extract relevant information.

The details on the available methods and additional subclasses implemented in MSPICE are described in detail in Section 5,Section 6 and Section 7.

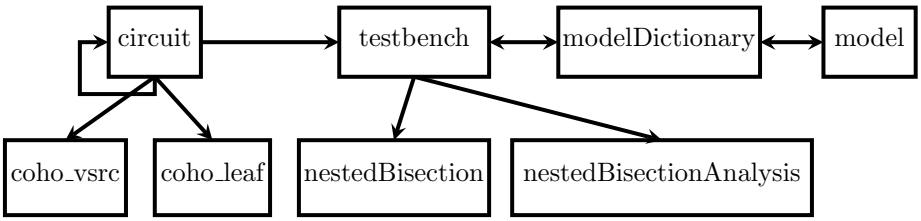

Figure 1: MSPICE software structure

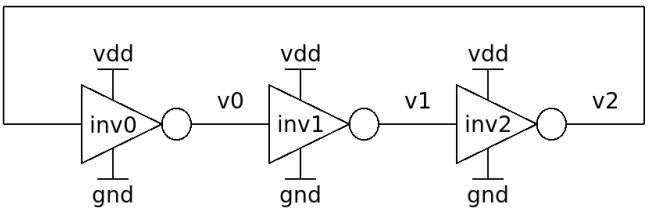

Figure 2: Three Stage Ring Oscillator

1.3 Citing MSPICE

We ask that if you use this code base for your own that you use the following citation:

2 MSPICE: Quick Start Guide Three Stage Ring Oscillator

Consider a simple three stage ring oscillator shown in Figure 2 to illustrate creating a circuit, a testbench, and running a simulation. This quick start guide is intended to distil the features available to demonstrate how to build a circuit in MSPICE (the three stage ring oscillator), create a testbench and run a simulation. Section 6 provides a comprehensive list of features and tools available for the user.

In this guide I will show you how to define the three stage ring oscillator from a template file by first defining the inverter `INV4.,` component and then the three stage ring oscillator `THREE_RING_OSC.m`. Section 2.2 describes the details on setting up a `testbench` with the three stage ring oscillator and how to obtain the results shown in Figure 3.
The inverter component definition is found in:

```
./ccm/libs/coho/components/INV4.m
```

The three stage ring oscillator is found in:

```
./ccm/libs/coho/circuits/THREE_RING_OSC.m
```

And the script to run the example is found in:

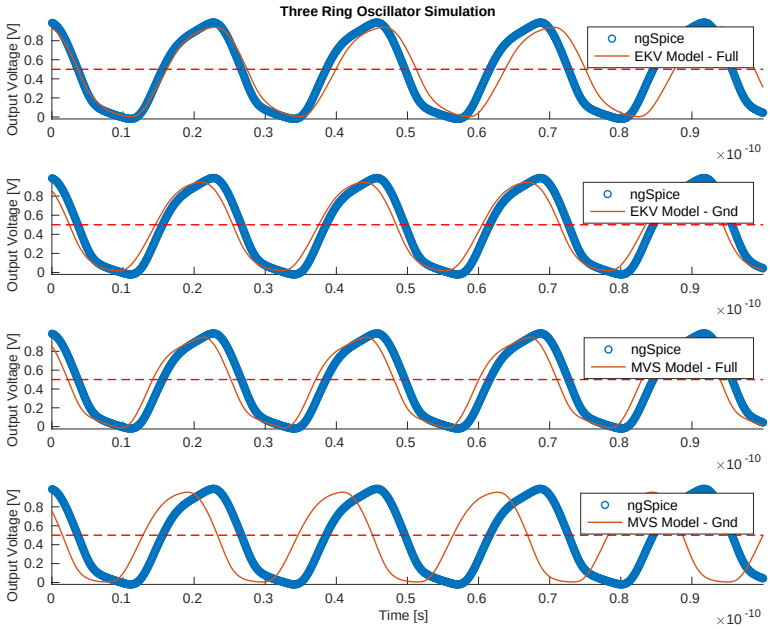

Figure 3: Three Stage Ring Oscillator model comparisons

`./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`

The tutorial starts with defining the circuit definition for `INV4.m` and demonstrates how once we have a component definition this can be used to compose larger circuits like the three stage ring oscillator shown in Figure 2. You can open these files and follow along in the tutorial to understand how the individual piece work instead of re-creating them from the `TEMPLATE.m` file.

Opening and running the script found in:
`./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`
will produce the results found in Figure 3. If you do not have Rump's IntLAB Automatic Differentiation (AD) package [18], then comment out line 26 of the file:

```
1 [tMvsFull,VmvsFull]    = tb_mvsFull.simulate([0  2e
      -10],[0,0,0,1,0]);
```

and comment out lines 55 to 64:

```
1 subplot(4,1,3)
2 hold on
3 plot(timeSim,Vo3,'o')
4 plot(tMvsFull-1.8e-11-1.1e-12,VmvsFull(:,5))
5 plot(tMvsFull,0.5*ones(1,length(tMvsFull)),'r--')
6 ylabel('Output Voltage [V]')
7 legend('ngSpice','MVS Model - Full')
```

```
set(gca,'fontsize',18)
axis([0,timeSim(end),-inf,inf])
set(findall(gca,'type','line'),'linewidth',1.2)
```

There will be a missing subplot after the script is completes running, however you will get an idea of the differences between the models considered.

2.1 Defining the circuits

To define the three stage ring oscillator, open the circuit template file found in:

`./ccm/libs/coho/circuits/TEMPLATE.m`

This template file has all the necessary scaffolding to define your own circuit and should look like:

```
% Template to be used for your own circuit
classdef TEMPLATE < circuit

    properties (GetAccess = 'public', SetAccess = '
        private')
        %Input; %outputs
    end

    methods
        function this = TEMPLATE(name,wid,rlen)
            if(nargin<2||isempty(name)||isempty(wid))
                error('must provide name and width');
            end
            if(nargin<3||isempty(rlen)), rlen = 1; end
            if(numel(wid)==1)
            else
                assert(length(wid) == 0) %replace 0 with
                    the correct number
            end
            this = this@circuit(name);

            this.finalize;

        end

    end
end
```

Notice that the three stage ring oscillator is made of 3 inverters, so let's use the above template twice. Once to define a simple inverter circuit and a second time to define the three stage ring oscillator. At the end of this portion of the guide you should have circuit definitions that look like the already defined circuits `INV4.m` and `THREE_RING_OSC.m` found in `./ccm/libs/coho/components/INV4.m` and `./ccm/libs/coho/circuits/THREE_RING_OSC.m` respectively.

2.1.1 Inverter `INV4.m`

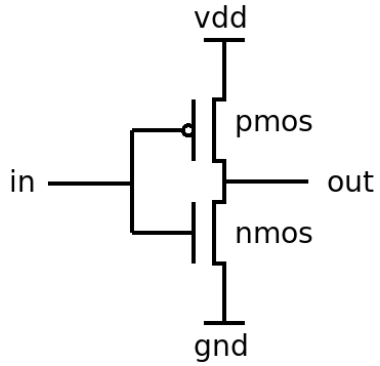

Figure 4: Inverter circuit

An inverter is made of two transistors as shown in Figure 4. The following list of instructions show how to fill in the template file for the definition of an inverter circuit explaining the methods along the way. At the end of following these instructions you will have written the definition for the inverter circuit found in `./ccm/libs/coho/circuits/INV4.m`.

You can skip reading these instruction and look at the final version of the code listing found at the end of this section. The instruction provides detailed explanations on what each line of code is doing with regards to building the circuit definition and will be assumed to be known when we continue to define the three stage ring oscillator in Section 2.1.2.
The following steps are instruction on defining `INV4.m`:

1. Add some documentation to explain what this circuit is/does and details on the constructor as shown:

```
1 % This class defines the 'inverter' circuit for COHO
    library
2 % An 'inverter' has two nodes: i(input) and o(output)
3 % To create an inveter, need to provide
4 %   1. name: circuit name
5 %   2. wid:  circuit width
6 %            wid(1) is the nmos width (nmos)
7 %            wid(2) is the pmos width (pmos)
8 %   3. rlen: (circuit length)/(minimum length), use 1
    by default
9 % E.g. inv = INV('inv',[1e-5,2e-5],1)
10 % NOTE: It's different from 'inverter' that 'INV'
    consists of 'nmos' and 'pmos'.
11 %  i.e. a function call of 'INV' requires two
    function calls from 'nmos','pmos'.
12 %  while a function call of 'inverter' issues one
    call from the mat file directly.
```

2. Then replace the keyword `TEMPLATE` in lines 2 and 9 (of the template file Listing 2.1) with the name of the circuit: in this case `INV4`. (There already exists an INV element for 3 terminal transistors - legacy of coho v1)

```
1 classdef INV4 < circuit
```

```
1     function this = INV4(name,wid,rlen)
```

3. Save the file as `INV4.m`, this is required so that MATLAB can properly call the constructor of `INV4`.

4. Now we add some properties to the circuit, for the inverter we have as inputs `vdd`, `gnd`, `i`, and we have the output `o`. We add these under the properties of line 5 of the template, as:

```
1   properties (GetAccess='public', SetAccess='private'
        );
2     vdd,gnd,i; o;
```

5. To allow for different invocations of inverters we add some guards to throw errors and provide default values where appropriate:

```
1       if(nargin<2||isempty(name)||isempty(wid))
2         error('must provide name and width');
3       end
```

The above snippet will throw an error if we try to create an inverter without a name and width.

6. The convention used in MSPICE is that if a single value for the width is provided in the constructor, it applies that width to all transistors in the circuit definition. Otherwise we require the length of *wid* to be of the number of devices in the circuit definition to do this we add a guard as follows:

```
1       if(nargin<3||isempty(rlen)), rlen = 1; end
2       if(numel(wid)==1)
3           wid = [1;1]*wid;
4       else
5           assert(length(wid)==2);
6       end
```

Note that it is the user's responsibility to define the widths in *wid* in the correct order for the circuit definition. Also notice that if *rlen* is not defined, by default it is set to 1. The parameter *rlen* allows the user to modify the length of the device. In all the examples provided here it is set to 1.

7. Finally we now invoke the constructor of the `circuit` class and assign it the variable *this* as shown:

```
1       this     = this@circuit(name);
```

By following the above instruction our `TEMPLATE` file should look like:

```
% This class defines the 'inverter' circuit for COHO library
% An 'inverter' has two nodes: i(input) and o(output)
% To create an inveter, need to provide
%   1. name: circuit name
%   2. wid:  circuit width
%            wid(1) is the nmos width (nmos)
%            wid(2) is the pmos width (pmos)
%   3. rlen: (circuit length)/(minimum length), use 1 by default
% E.g. inv = INV('inv',[1e-5,2e-5],1)
% NOTE: It's different from 'inverter' that 'INV' consists of 'nmos' and 'pmos'.
%  i.e. a function call of 'INV' requires two function calls from 'nmos','pmos'.
%  while a function call of 'inverter' issues one call from the mat file directly.
classdef INV4 < circuit
  properties (GetAccess='public', SetAccess='private');
    vdd,gnd,i; o;
  end
  methods
    % wid: wid(1) is the width of nmos, and wid(2) is the width of pmos
    %      if wid(2) is not provided, 2*wid(1) is used by default
    % rlen: relative transitor length, must be scaler
    function this = INV4 (name,wid,rlen)
      if(nargin<2||isempty(name)||isempty(wid))
        error('must provide name and width');
      end
      if(nargin<3||isempty(rlen)), rlen = 1; end
      if(numel(wid)==1)
          wid = [1;1]*wid;
      else
          assert(length(wid)==2);
      end

      this     = this@circuit(name);
```

We can now define circuit nodes, and make connections to define our circuit.

1. First we need to add nodes and ports to the circuit by creating nodes via `node` and adding ports via `this.add_port(`*`node_object`*`)`. These can be nested. In the inverter example we do the following to add our inputs and outputs to the circuit:

```
        this.vdd = this.add_port(node('vdd'));
        this.gnd = this.add_port(node('gnd'));
        this.i   = this.add_port(node('i'));
```

```
4        this.o   = this.add_port(node('o'));
```

The above creates nodes and ports *vdd*, *gnd*, *i*, *o* for the circuit. A circuit port requires a `node` object.

2. We then create the devices which make up the inverter and distribute the widths to their appropriate location. An inverter is made of one `nmos` device and one `pmos` device. To add these devices to our circuit we do the following:

```
1        widN = wid(1);
2        widP = wid(2);
3
4        n = nmos('n','wid',widN,'rlen',rlen); this.
             add_element(n);
5        p = pmos('p','wid',widP,'rlen',rlen); this.
             add_element(p);
```

Each transistor device has nodes *d* (drain), *g* (gate), *s* (source), *b* (body) which need to be connected. Note also that after a device is created e.g. `n = nmos('n','wid',widN,'rlen',rlen)` it needs to be added to the circuit by calling `this.add_element(device)`. Transistor devices are created using the MATLAB convention of `<'string',value>` pairs. As shown in the above snippet in lines 4 and 5. The constructor of a `(p/n)mos` device has the following signature:

```
(p/n)mos(name,<wid,wid_value>,<rlen,rlen_value>)
```

3. Once all the devices have been created, we proceed to connecting the device nodes to the nodes within our circuit. Connections are made by calling `this.connect(`$node_1$`,`$node_2$`,`$\cdots$`,`$node_n$`)`, which connect all nodes$_{1,2,\cdots,n}$ together. The connections in the inverter circuit are defined as follows:

```
1        this.connect(this.i,n.g,p.g);
2        this.connect(this.o,n.d,p.d);
3        this.connect(this.vdd,p.s,p.b);
4        this.connect(this.gnd,n.s,n.b);
```

The input node *this.i* is connected to `pmos` and `nmos` nodes *g* and so on.

4. To finalize the circuit to be used in a testbench, the circuit class method `this.finalize` needs to be called as shown in line 20 of the `TEMPLATE.m` file.

```
1        this.finalize;
```

Having followed the above steps, the circuit definition for `INV4.m` should look as follows:

```
1 % This class defines the 'inverter' circuit for COHO
    library
2 % An 'inverter' has two nodes: i(input) and o(output)
```

```
3  % To create an inveter, need to provide
4  %   1. name: circuit name
5  %   2. wid:  circuit width
6  %            wid(1) is the nmos width (nmos)
7  %            wid(2) is the pmos width (pmos)
8  %   3. rlen: (circuit length)/(minimum length), use 1 by
       default
9  % E.g. inv = INV('inv',[1e-5,2e-5],1)
10 % NOTE: It's different from 'inverter' that 'INV'
       consists of 'nmos' and 'pmos'.
11 %  i.e. a function call of 'INV' requires two function
       calls from 'nmos','pmos'.
12 %  while a function call of 'inverter' issues one call
       from the mat file directly.
13 classdef INV4 < circuit
14   properties (GetAccess='public', SetAccess='private');
15     vdd,gnd,i; o;
16   end
17   methods
18     % wid: wid(1) is the width of nmos, and wid(2) is the
           width of pmos
19     %      if wid(2) is not provided, 2*wid(1) is used by
           default
20     % rlen: relative transitor length, must be scaler
21     function this = INV4 (name,wid,rlen)
22       if(nargin<2||isempty(name)||isempty(wid))
23         error('must provide name and width');
24       end
25       if(nargin<3||isempty(rlen)), rlen = 1; end
26       if(numel(wid)==1)
27           wid = [1;1]*wid;
28       else
29           assert(length(wid)==2);
30       end
31
32       this     = this@circuit(name);
33
34       this.vdd = this.add_port(node('vdd'));
35       this.gnd = this.add_port(node('gnd'));
36       this.i   = this.add_port(node('i'));
37       this.o   = this.add_port(node('o'));
38
39       %create devices wid is of the form wid = [widN,widP
             ] if wid is
40       %called as a single number then both the P and N
             device will
41       %have the same width.
42
43       widN = wid(1);
44       widP = wid(2);
```

```
45
46          n = nmos('n','wid',widN,'rlen',rlen); this.
                add_element(n);
47          p = pmos('p','wid',widP,'rlen',rlen); this.
                add_element(p);
48
49          this.connect(this.i,n.g,p.g);
50          this.connect(this.o,n.d,p.d);
51          this.connect(this.vdd,p.s,p.b);
52          this.connect(this.gnd,n.s,n.b);
53          this.finalize;
54        end
55    end
56  end
```

2.1.2 Three stage ring oscillator: `THREE_RING_OSC.m`

With the `INV4.m` circuit definition complete we can now use it to compose other circuits. We finish the circuit definition part of the quick start guide by filling in a new `TEMPLATE.m` file so that we end up with the circuit definition found in `./ccm/libs/coho/circuits/THREE_RING_OSC.m`. The following steps gets us to the final circuit definition which will then be used to create a testbench and simulate the circuit using different models and settings in Section 2.2. Building on the knowledge from defining the `INV4.m` circuit definition, we define `THREE_RING_OSC.m` by:

1. Replace lines 2 and 9 of `TEMPLATE.m` with `THREE_RING_OSC`.
2. Add properties *vdd* and *gnd* to line 5.
3. At line 14 add:

```
1                   widInv0=[1;1]*wid;
2                   widInv1=[1;1]*wid;
3                   widInv2=[1;1]*wid;
```

4. Change line 16 with:

```
1                   assert(length(wid) == 6)
2                   widInv0 = wid(1:2);
3                   widInv1 = wid(3:4);
4                   widInv2  = wid(5:end);
```

5. Add after line 18 of the `TEMPLATE.m`: `this = this@circuit(name);` what to do with *rlen*:

```
1               rlen0 = rlen;
2               rlen1 = rlen;
3               rlen2 = rlen;
4               if(iscell(rlen))
5                   rlen0 = rlen{1};
6                   rlen1 = rlen{2};
```

```
7                   rlen2 = rlen{3};
8               end
```

6. Add ports *vdd*, and *gnd* as:

```
1               this.vdd = this.add_port(node('vdd'));
2               this.gnd = this.add_port(node('gnd'));
```

7. Add internal nodes *v0*, *v1*, *v2* and `INV4.m` devices *inv0*, *inv1* and *inv2* as:

```
1               v0 = this.add_port(node('v0'));
2               v1 = this.add_port(node('v1'));
3               v2 = this.add_port(node('v2'));
4
5               inv0 = INV4('inv0',widInv0,rlen0); this.
                    add_element(inv0);
6               inv1 = INV4('inv1',widInv1,rlen1); this.
                    add_element(inv1);
7               inv2 = INV4('inv2',widInv2,rlen2); this.
                    add_element(inv2);
```

8. Next we connect the output of *inv2* to node *v2* and to the input of *inv1*:

```
1               this.connect(v2,inv2.o,inv0.i);
```

   connect the output of *inv0* to node *v0* and the input to *inv1*:

```
1               this.connect(v0,inv0.o,inv1.i);
```

   connect the output of *inv1* to node *v1* and the input to *inv2*:

```
1               this.connect(v1,inv1.o,inv2.i);
```

   and lastly we need to connect *vdd* and *gnd* to power the inverters:

```
1               this.connect(this.vdd,inv0.vdd,inv1.vdd,
                    inv2.vdd);
2               this.connect(this.gnd,inv0.gnd,inv1.gnd,
                    inv2.gnd);
```

9. The last step is to finalize the circuit which is in line 20 of `TEMPLATE.m`.

With the appropriate documentation your version of the three stage ring oscillator circuit should look like:

```
1 % Definition of a THREE_RING_OSC a three stage ring
    oscillator with
2 % INV4 elements
3 %
4 % The three ring oscillator has 2 input nodes:
5 % 1. vdd:    power supply source
6 % 2. gnd:    circuit ground
7 % The three ring oscillator has 3 output nodes:
8 % 1. v1:     the first output of the oscillator
```

```
% 2. v2:     the second output of the oscillator
% 3. v3:     the third output of the oscillator
%
% To create a THREE_RING_OSC, requires
% 1. name: oscillator name
% 2. wid: circuit width
%       - if one wid is given that width is applied to all devices
%       - otherwise width needs to be of length 6:
%           wid(1:2) = is for inv0 (INV4)
%           wid(3:4) = is for inv1 (INV4)
%           wid(5:6) = is for inv2 (INV4)
% 3. rlen: relative device length, use 1 by default.
% E.g. myOsc = THREE_RING_OSC('osc0',450e-7,1)
classdef THREE_RING_OSC < circuit

    properties (GetAccess = 'public', SetAccess = 'private')
        vdd,gnd; %Input
    end

    methods
        function this = THREE_RING_OSC(name,wid,rlen)
            if(nargin<2||isempty(name)||isempty(wid))
                error('must provide name and width');
            end
            if(nargin<3||isempty(rlen)), rlen = 1; end
            if(numel(wid)==1)
                widInv0=[1;1]*wid;
                widInv1=[1;1]*wid;
                widInv2=[1;1]*wid;
            else
                assert(length(wid) == 6)
                widInv0 = wid(1:2);
                widInv1 = wid(3:4);
                widInv2  = wid(5:end);
            end
            this = this@circuit(name);
            %set the relative lengths by default to be be the same
            rlen0 = rlen;
            rlen1 = rlen;
            rlen2 = rlen;
            if(iscell(rlen))
                rlen0 = rlen{1};
                rlen1 = rlen{2};
                rlen2 = rlen{3};
            end
            %define circuit nodes
            %inputs
```

```
56              this.vdd = this.add_port(node('vdd'));
57              this.gnd = this.add_port(node('gnd'));
58
59              %internal node/outputs
60              v0 = this.add_port(node('v0'));
61              v1 = this.add_port(node('v1'));
62              v2 = this.add_port(node('v2'));
63
64              inv0 = INV4('inv0',widInv0,rlen0); this.
                    add_element(inv0);
65              inv1 = INV4('inv1',widInv1,rlen1); this.
                    add_element(inv1);
66              inv2 = INV4('inv2',widInv2,rlen2); this.
                    add_element(inv2);
67
68              this.connect(v2,inv2.o,inv0.i);
69              this.connect(v0,inv0.o,inv1.i);
70              this.connect(v1,inv1.o,inv2.i);
71              this.connect(this.vdd,inv0.vdd,inv1.vdd,inv2.
                    vdd);
72              this.connect(this.gnd,inv0.gnd,inv1.gnd,inv2.
                    gnd);
73
74
75              this.finalize;
76
77          end
78
79      end
80  end
```

2.2 Simulating the Three Stage Ring Oscillator

With the completed circuit definition, we are now ready to use it in a `testbench`. The `testbench.m` class is the backbone of the MSPICE simulator. The signature for creating a `testbench` is as follows:

`testbench(`*`circuit`*`,`*`ports`*`,`*`sources`*`,`*`model`*`,`*`process`*`,`*`options`*`)`
Creating a `testbench` requires:

1. A *`circuit`* definition such as the one described in Section 2.1.2.
2. The *`circuit`*'s *`ports`* are an ordered cell array, i.e. {`myOsc.vdd,myOsc.gnd`}.
3. The *`sources`* are external sources that connect to the *`circuit`*'s *`ports`* in the order they appear as a cell array, i.e. {`Vdd,Gnd`}.
4. The *`model`* is the default `model` that the `testbench` uses for `simulation`.
5. The *`process`* is the `process` for the `testbench`'s *`model`*

6. The *options* is the `option` object that determine parameters for the `testbench` at simulation time.

The *circuit* we will use is the `THREE_RING_OSC.m` circuit which we defined in Section 2.1.2. The *ports* of `THREE_RING_OSC.m` are *vdd* and *gnd*, but we need to create voltage sources for the `testbench` to connect to the ports *vdd* and *gnd* of the `THREE_RING_OSC.m`. Once the voltage sources are created, we make a `testbench` option object to setup the `testbench` with its circuit and voltage sources. Finally we select our `model` and `process`. Once the `testbench` is created we can use the `simulate` method to run our simulation. A full list of methods for the `testbench` class is found in Section 6.

2.2.1 Creating voltage sources

There are a number of voltage sources available in MSPICE, a full list is found in Section 5.4. Here we consider only those that are constant voltage sources used to power the `THREE_RING_OSC.m` circuit which are created by calling

`vsrc('name',value,ctf)`

where '*name*' is the name of the voltage source, *value* is the DC voltage value the source outputs, and *ctg* (connect-to-ground) is a flag which if true connects the negative terminal of the voltage source to ground. To create the necessary sources for the *vdd* and *gnd* ports of our `THREE_RING_OSC.m` circuit the following lines of code in `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m` define these voltage sources:

```
Vdd = vsrc('vdd',1.0,1);
Gnd = vsrc('gnd',0.0,1);
```

2.2.2 Creating the `circuit`

To create a circuit object which is passed into the `testbench` we simply instantiate our circuit definition as:

```
ringOsc = THREE_RING_OSC('osc',450e-7,1);
```

which creates our three stage ring oscillator with all transistor widths set to $450nm$.

2.2.3 Creating the `testbench` options

All the `testbench` settings that are available are outlined in Section 6.1.1. The settings used to setup the testbenches in `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m` are as follows:

```
tbOptionsFull = struct('capModel','full','capScale',1,'
    vdd',1,'temp',298,'numParallelCCTs',1,'debug',false);
tbOptionsGnd  = struct('capModel','gnd','capScale',1,'vdd
    ',1,'temp',298,'numParallelCCTs',1,'debug',false);
```

the important difference between these settings is in the capacitance models employed:

1. `<'capModel','gnd'>`
2. `<'capModel','full'>`

The difference between these two capacitance models for the MOS devices within the test bench is weather inter-node capacitance which models effects such as Miller capacitance is considered. The capacitance model `<'capModel','gnd'>` only considers capacitances with respect to ground, while `<'capModel','full'>` considers internet node capacitances where applicable.

If you do not have Rump's AD package [18] comment out line 26:

```
1 [tMvsFull,VmvsFull]    = tb_mvsFull.simulate([0  2e
      -10],[0,0,0,1,0]);
```

The `testbench` options are structured in the MATLAB convention of `<'string',value>` pairs stored in a MATLAB `struct` object. Shown in the setup for the `THREE_RING_OSC.m` circuit are the following pairs:

1. `<'capModel','gnd/full'>`, the options are *gnd* or *full*, where *gnd* defines capacitances with respect to ground and *full* defines capacitance with respect to ground and inter-node.
2. `<'capScale',value>`, *value* scales $\frac{dV}{dt}$ by *value* for better scaling of numerical integration.
3. `<'vdd',value>`, *value* is the global value for the circuit's *vdd*, legacy of v1 of coho where needed for auto-connection. In the nested bisection algorithm (see Section 6.2), it is useful to have knowledge of *vdd* of the circuit.
4. `<'temp',value>`, *value* is the temperature the circuit is running at in degrees Kelvin.
5. `<'numParallelCCTs',value>`, *value* is the number of circuits being simulated in parallel. This feature allows the user to simulate multiple copies of the circuit with differing initial conditions - a necessary feature for the nested bisection algorithm described in Section 6.2
6. `<'debug',true/false>`, the *debug* flag turns on or off debug mode of the `testbench` . When *debug* mode is *true*, all numerically integrated data points and intermediate calculations such as *Vin*, *I* and *C* are saved in a *debug* cell array.

2.2.4 Creating the `testbench`

In `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`, each `testbench` is created as shown in lines 18 through 22:

```
1 tb_ekvFull = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
      },{Vdd,Gnd},'EKV','PTM 45nmHP',tbOptionsFull);
2 tb_ekvGnd  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
      },{Vdd,Gnd},'EKV','PTM 45nmHP',tbOptionsGnd);
3
4 tb_mvsFull  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
      },{Vdd,Gnd},'MVS','PTM 45nmHP',tbOptionsFull);
```

```
5 tb_mvsGnd  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
    },{Vdd,Gnd},'MVS','PTM 45nmHP',tbOptionsGnd);
```

The `THREE_RING_OSC.m` ports *vdd* and *gnd* correspond to the order in which the voltage sources *Vdd* and *Gnd* appear. These are passed into the `testbench` constructor as cell arrays. The *model* and *process* parameters are passed in as strings and finally the *options* is a `struct` as defined in Section 2.2.3.

In this example, I compare the results obtained by simulating `THREE_RING_OSC.m` using my simplified *EKV* model described in [17] and the adapted *MVS* model [14] which have been fit to data obtained using ngSPICE [19] and the PTM $45nm$HP model card. The results in Figure 3 show the differences in these two models when using a fully connected capacitance model vs capacitance referenced to ground only as compared to what ngSPICE produces.

The `testbench` parameters *model* and *process* signal to the `modelDictionary.m` class (see Section 7) which *model* and *process* to use. Here these are *EKV* and *MVS* models and the *PTM 45nmHP* process.

2.2.5 Simulating `THREE_RING_OSC.m`

Once the `testbench` is created such as `tb_ekvFull` as shown in Section 2.2.4 then to obtain a time series result a call to the `simulate` method invokes the `testbench`'s integrator to simulate the `circuit` defined in the `testbench` over the specified time interval, i.e:

```
myTestbench.simulate([starTime,stopTime],v0,options)
```

The parameters *v0* and *options* are optional in this case and can be used to specify initial conditions and integrator options respectively. In `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`, the simulations are defined as:

```
1 [tEkvFull,VekvFull] = tb_ekvFull.simulate([0 2e
    -10],[0,0,0,1,0]);
2 [tEkvGnd,VekvGnd]   = tb_ekvGnd.simulate([0 2e
    -10],[0,0,0,1,0]);
3 [tMvsFull,VmvsFull]   = tb_mvsFull.simulate([0 2e
    -10],[0,0,0,1,0]);
4 [tMvsGnd,VmvsGnd]   = tb_mvsGnd.simulate([0 2e
    -10],[0,0,0,1,0]);
```

where *startTime* is 0 and *stopTime* is 2×10^{-10} seconds. The initial conditions *v0* specifies the starting condition for the voltage sources and the internal circuit nodes *v0*, *v1* and *v2*. In this case *v0* starts at 0, *v1* starts at 1 and *v2* starts at 0. Notice that no integrator options are specified, thus internal defaults are selected. The results of the simulation are shown in Figure 3 at the beginning of this section.

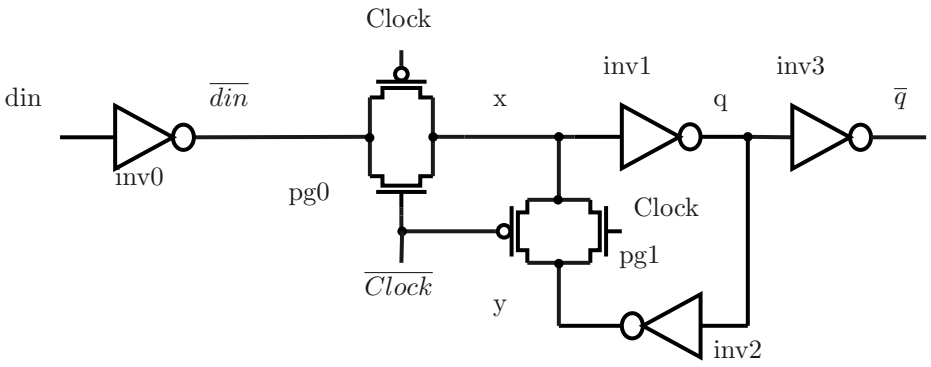

Figure 5: Passgate Latch Circuit

3 Synchronizer Metastability Simulation and Analysis Guide: Passgate Latch Synchronizer

This guide is intended to describe the components found in:

```
./ccm/Examples/Synchronizer Analysis/Passgate Latch/singlePGLatchRun.m
```

and runs through the use of the `nestedBisection.m` and `nestedBisectionAnalysis.m` `testbench` subclasses on a passgate latch synchronizer shown in Figure 5. Details on the algorithm and its analysis can be found in [15–17, 22, 23]. Figure 6 summarizes the results of running the example script.

There are several details to pay attention to in order to simulate metastability behaviour in a synchronizer and its subsequent analysis. It is assumed that the synchronizer has at least one clock source and is parametrized with an input source that will result in the synchronizer's final output to be different for at least two different input values to the parameterized input source.

The analysis requires the user to select a metric used to quantify the progress that the synchronizer has made towards metastability resolution. In this example this is the difference between nodes x and q. In addition to a user selected metric, the user needs to provide a critical time t_{crit} at which point the synchronizer circuit will just meet the digital behaviour criteria.

Furthermore functions `stopToDigitalSim.m` and `nestedBisectionStop.m` need to be defined to signal when a nested bisection round ends, and when to stop the algorithm respectively.

The circuit used in this analysis can be found in:

```
./ccm/libs/coho/circuits/PASSGATE_LATCH_TEST.m
```

Which defines the circuit shown in Figure 5. Its circuit definition is made of components `PASSGATE_LATCH.m` and `INV4.m`. More details on how to make a circuit definition can be found in Section 2.

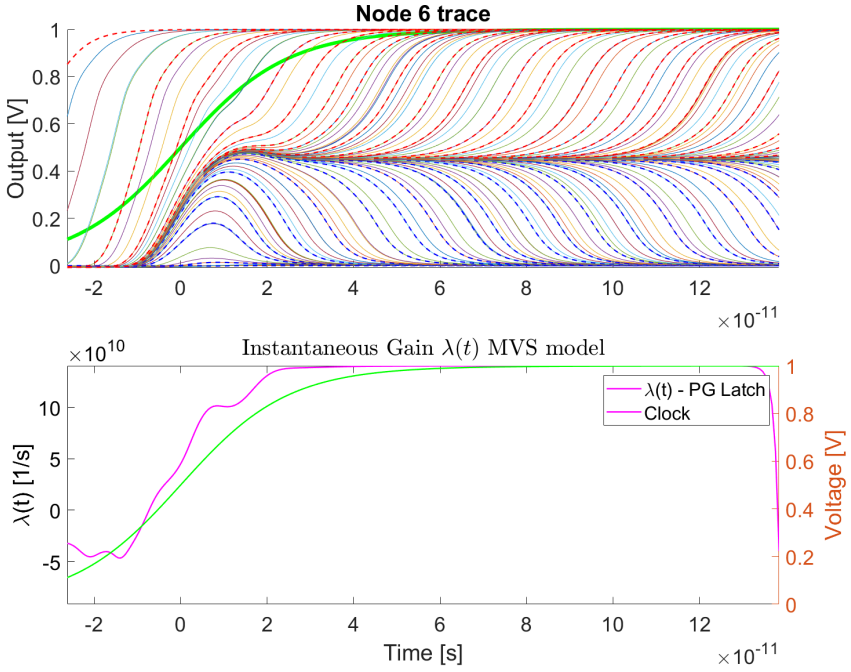

Figure 6: Nested Bisection and Analysis Results - Passgate Latch

3.1 Creating the voltage sources

The guide starts with the particular voltage sources required for the `nestedBisection testbench`. The voltage sources created for the passgate latch synchronizer are shown in lines 1 through 7:

```
1 %voltage sources
2 gnd = vsrc('gnd',0,1);
3 vdd = vsrc('vdd',1,1);
4 clk = vtanhClock('clk',1e10,-pi/2,0.5,0.5,4,3e-10,0,1);
5 clkbar = vtanhClock('clkbar',1e10,pi/2,0.5,0.5,4,3e
      -10,1,1);
6
7 din = vtanhDelay('din',0.5,0.5,5e10,true);
```

The `vsrc` voltage source is covered in Section 2.2 (as well as the list of available voltage sources in Section 5.4). New here is the voltage sources `vtanhClock` and `vtanhDelay`.

The `vtanhClock` source is periodic and used to simulate the synchronizer's clock signal (and where necessary its logical inverse). Whereas `vtanhDelay` is parameterized by a variable *delay* which is necessary for the nested bisection algorithm.

3.2 The `nestedBisection testbench`

The details on creating a **`nestedBisection testbench`** are found in Section 6.2. Line 26 creates the **`nestedBisection testbench`**.

```
1 nbPGLatch = nestedBisection(pg,{pg.vdd,pg.gnd,pg.clk,pg.
      clkbar,pg.d},...
2       {vdd,gnd,clk,clkbar,vdd},[5.5e-10 4e-10],{pg.d,din},
          clk,mask,'EKV','PTM 45nmHP');
```

The previous section describes the various voltage sources used for the synchronizer, line 16 defines the ***`mask`*** for the **`testbench`** :

```
1 mask = [pg.cctPath.q,pg.cctPath.q];
```

The **`circuit`** class constructs an object **`cctPath`** which allows the user to navigate the circuit definition nodes.

The first node of the *mask* is not used in determining metastability trajectories and meant to be used to observe any node of the circuit. In this example the output node q is the only node of interest, thus it is repeated twice.

The ***`model`*** and ***`process`*** are defined in the same way as a **`testbench`**.

In addition to specifying the above parameters for a **`nestedBisection testbench`**, the functions **`stopToDigitalSim.m`** and **`nestedBisectionStop.m`** need to be defined. These are user provided functions which determine the condition upon which to stop a nested bisection round and the condition upon which to terminate the nested bisection algorithm respectively.

3.2.1 Defining `stopToDigitalSim.m`

Details on this function can be found in Section 6.2.4. The implementation for the passgate latch synchronizer example is as follows:

```
1  function [value,isterminal,direction] = stopToDigitalSim
       (t,Vin,vDot,digitalSimOptions)
2          numOfNodes = length(Vin);
3          vdd = digitalSimOptions.vdd;
4          minSimTime = digitalSimOptions.minSimTime;
5          threshold = digitalSimOptions.threshold;
6          stopGain = digitalSimOptions.stopGain;
7
8          stopCondition = sum(tanh(stopGain*(vdd*(1-
               threshold)*ones(1,numOfNodes)*1.05-vdd/2).^2
               ...
9            -(threshold - vdd/2)^2));
10
11         if(t == minSimTime)
12              t = minSimTime + 1e-6;
13         end
14
15         a = stopGain*sign(t-minSimTime);
16         isterminal = 1;
17         direction = 1;
18         value = sum(tanh(a*(Vin-vdd/2).^2 -...
```

```
19                  (threshold-vdd/2)^2))-stopCondition;
20      end
```

The specific digital simulation options are unpacked from `digitalSimOptions` and the `stopCondition` in this example is when all voltage nodes are within `threshold` of `gnd` or `vdd`. In addition we only allow the simulation to stop after a minimum simulation time `minSimTime` is elapsed. This prevents the stopping condition from being triggered before t_{in} occurs to change the synchronizer state. When all voltage states of the circuit are within `threshold` of `gnd` or `vdd`, then `value` becomes positive, which terminates that round of the nested bisection algorithm.

A safety margin of 5% is added to prevent erroneously stopping early, thus the entire voltage state needs to get to within the defined `threshold` + 5% before value changes direction from negative to positive causing the integration to stop.

$$\begin{aligned} stopCond &= \sum_{i=1}^{N} \tanh\left[(g(1.05V_{dd}(1-threshold))\mathbf{u} - V_{dd}/2)^2 - (threshold - V_{dd}/2)^2\right] \\ a &= g \cdot \text{sign}(t - t_{minTime}) \\ value &= \sum_{i=1}^{N} \tanh\left[(a(\mathbf{V}_{in} - V_{dd}/2)^2 - (threshold - V_{dd}/2)^2)\right] - stopCond \end{aligned} \tag{1}$$

g is a scaling factor to make `value` change more sharply, $\mathbf{u}$ is a vector of ones of the same length as $\mathbf{V}_{in}$, and a ensures that `value` is negative for time $t \in [0, t_{minTime}]$.

3.2.2 Defining `nestedBisectionStop.m`

Details on this function can be found in Section 6.2.5. The implementation in the passgate latch synchronizer is as follows:

```
1  function stop = nestedBisectionStop(tbOptions,
       numStatesPerCCT,Vin,tCrit,t)
2  %nestedBisectionStop is a function which determines the
       stopping condition upon
3  %which to terminate the nested bisection algorithm. For
       example when the
4  %before last stage of the synchronizer has resolved
       metastability by
5  %tCrit. When this function gets called the simulation
       time has exceeded the
6  %user specified tCrit time and this function evaluates
       the selected
7  %criteria to return TRUE if the nested bisection
       alogirthm is to halt,
8  %otherwise returns FALSE.
9  %
10 %Inputs: numParallelCCTs - the number of parallel
       circuits being simulated.
11 %         numStatesPerCCT - the number of states per
       circuit
```

```
12 %         Vin              - the integrated voltage state
       vector
13 %         tCrit            - the critical time
14 %         t                - the sim time
15
16 numParallelCCTs = tbOptions.numParallelCCTs;
17 railLimit = tbOptions.digitalSimOptions.threshold;
18 vdd = tbOptions.vdd;
19 failHigh = (1-railLimit)*vdd;
20 failLow  = railLimit*vdd;
21
22 metaResolveNode = 1;
23
24 stop = false;
25 hCond= false;
26 lCond = false;
27 ind = find(t>= tCrit);
28 ind = ind(1);
29
30 for i = 1:numParallelCCTs
31     ViQmeta = Vin(ind,metaResolveNode + (i-1)*
          numStatesPerCCT);
32     ViQEndmeta = Vin(end,metaResolveNode + (i-1)*
          numStatesPerCCT);
33     if( (ViQmeta < failHigh) && (ViQEndmeta > failHigh))
34         hCond = true;
35     end
36     if( (ViQmeta > failLow) && (ViQEndmeta < failLow))
37         lCond = true;
38     end
39 end
40
41 if( hCond && lCond)
42     stop = true;
43 end
44 end
```

The `nestedBisection testbench options` object is passed in as a parameter to include information such as `vdd`, `numParallelCCTs` ect. The number of states per synchronizer circuit `numStatesPerCCT` gives the user the ability to segment the voltage state $\mathtt{V}_{in}$ into the appropriate number of simultaneously simulated circuits to search the trajectories and find if the relevant terminating criteria is met on the time interval $t \in [t_{crit}, t_{end}]$ (thus t and t_{crit} are passed in as parameters).

In this implementation the nested bisection algorithm is terminated if the output node q has a trajectory which starts off metastable at t_{crit} and ends in a digital *high* state and one where q has a trajectory which starts off metastable at t_{crit} and ends in a digital *low* state. This is enforced to collect enough data to describe and time shift trajectories to just meet synchronizer settling into digitally defined levels at time t_{crit}.

Internally to the nested bisection algorithm, `nestedBisectionStop` is invoked if the simulation end time t exceeds t_{crit}.

3.2.3 Running the nested bisection algorithm

The nested bisection algorithm is run by calling the `simulate` method as follows:

```
1 benchRuns = nbPGLatch.simulate('pgLatchEKV',span,tCrit);
```

which launches the nested bisection algorithm saving the data in `'pgLatchEKV.mat'`. The data structure is also returned as the variable `benchRuns`. This data is used in the analysis of the synchronizer. The time span is the span of time to simulate the synchronizer. The time span is internally shifted such that $t = 0$ occurs at the mid-point of the first edge of the clock signal.

Launching the nested bisection algorithm will display the nodes defined in the variable ***mask*** and provide a running numerical approximation to $\beta(t)$ (see [17] for details).

3.3 The `nestedBisectionAnalysis testbench`

The details on constructing a `nestedBisectionAnalysis testbench` are found in Section 6.4. In the running example, line 29 creates the testbench object:

```
1 nbPGLAnalysis = nestedBisectionAnalysis(pg,{pg.vdd,pg.gnd
      ,pg.clk,pg.clkbar,pg.d},...
2       {vdd,gnd,clk,clkbar,vdd},{pg.d,din},clk,mask(end),'
          EKV','PTM 45nmHP');
```

The main difference in constructing the analysis testbench and the nested bisection testbench is that the clock and initial time interval are not required but the output node/nodes of interest for determining the settling criteria at t_{crit} is provided. In this example the last node of the ***mask*** variable is used.

3.3.1 Running the analysis algorithm

Similar to the `nestedBisection testbench` the analysis is performed by invoking the `bisectionAnalysis` method, shown in line 35:

```
1 [tpgLatch,lambdaPGL] = nbPGLAnalysis.bisectionAnalysis('
      pgLatchAnalysisEKV',benchRuns,uVcritBench,tCrit);
```

The main observation to make here is that the variable `uVcritBench` is required. Which in this example is defined such that:

$$q - x \quad = \quad \texttt{uVcritBench} \cdot V(t) \tag{2}$$

Lines 18 to 21 define this variable:

```
1 uVcrit = [pg.cctPath.q,pg.cctPath.pgl0.x0];
2
3 uVcritBench = zeros(1,10);
4 uVcritBench(uVcrit) = [1,-1];
```

This variable selects from the voltage state of the synchronizer the effectiveness of the synchronizer towards metastability resolution. Similar to the `nestedBisection testbench`, the analysis data is saved in:

```
'pgLatchAnalysisEKV.mat'
```

In addition all raw Jacobian, $\frac{\partial f}{\partial t_{in}}$, $\dot{V}(t)$, $\frac{\partial I(t)}{\partial V(t)}$ and $C(V,t)$ is saved in:

```
'pgLatchAnalysisEKV_RawDerivativesAll.mat'
```

for additional analysis if required. The analysis returns the quantity $\lambda(t)$ and t, where $\lambda(t)$ is the portion of the solution to the differential equation:

$$\dot{g}(t) \quad = \quad \lambda(t)g(t) + \rho(t) \tag{3}$$

Extensive details are found in [15–17] for the description and interpretation of the results. The rest of the code plots and displays the results.

4 Optimizing Synchronizer

Section 3 describes `nestedBisection` and `nestedBisectionAnalysis` functionality for synchronizer metastability simulation and analysis. This section outlines the requirements to use the method `dGdw` from the `nestedBisectionAnalysis testbench` class. In addition there is a brief discussion on our approach to applying constraints described in [15] to provide some initial starting point in applying new user defined constraints.

The `nestedBisectionAnalysis` class has the ability to compute the gradient of the synchronizer gain $G_{sync} = g(t_{crit})$ with respect to its input transistor widths w. This can be used to perform a gradient descent optimization of a synchronizer design to maximize the synchronizer gain, thus maximizing it's effectiveness at metastability resolution. Computing $\frac{\partial G_{sync}}{\partial w}$ of a synchronizer requires that the nested bisection algorithm and analysis be complete. To give the user flexibility with the optimization, two user provided function `expandW.m` and `populateADoptimizationVar.m` are also required. Templates to these function can be found in:

```
./ccm/MSpice/expandW_Template.m
./ccm/MSpice/populateADoptimizationVar_Template.m
```

There are example scripts for optimizing two flip-flop passgate, jamb, robust jamb [24] and Strong Arm [10, 13] synchronizer designs in:

```
./ccm/Examples/Synchronizer Analysis/Two Flip-Flop Optimization
```

The jamb and robust jamb designs have additional scripts within their appropriate folders to impose further constraints.

The examples provided all impose that the overall width budget w_{tot} remains constant, that no individual transistor is smaller than a specified minimum width `minWidth` and where applicable the design is required to "work". The functions `expandW.m` and `populateADoptimizationVar.m` impose those

constraints however, the user is free to use these functions to apply their own constraints.

The optimization of the synchronizer designs in the provided examples is summarized as follows:

1. Run the nested bisection algorithm with widths w
2. Run the nested bisection analysis
3. Compute $\frac{\partial G_{sync}}{\partial w}$
4. Compute the derivative of the log of G_{sync} w.r.t. w because $g(t)$ grows exponentially: $\frac{\partial}{\partial w}\log(G_{sync}) = 1/g\frac{\partial G_{sync}}{\partial w}$
5. Apply the constraints by projecting the solution space to the hyperplane of constant width
6. Take a gradient decent step (step size according to the Wolfe condition)
7. Repeat analysis
8. Stop when the difference between two subsequent steps is small enough

4.1 `populateADoptimizationVar.m`

This function takes in the voltage state of the synchronizer and all device widths (*nDevices* and *pDevices*) and selects a subset of those devices to turn into a `hessian` object which is necessary to compute $\frac{\partial G_{sync}}{\partial w}$. The details on the function definition can be found in Section 6.4.14. The new AD variable takes on the following form:

$$adVar \quad = \quad \begin{bmatrix} V_{state} \\ w_{nDevices} \\ w_{pDevices} \end{bmatrix} \tag{4}$$

It is the responsibility of the user to determine how to appropriately distribute the device width such that they correspond to the appropriate devices within the circuit definition.

The MSPICE code groups devices by type, thus the connectivity matrix N which redistributes voltages and by extension device widths is important to keep in mind when constructing this function. This means that the gradient $\frac{\partial G_{sync}}{\partial w}$ is organized into a column vector by type and needs to be redistributed to correspond to the proper device definitions of the circuit. – automation of this step would be an improvement to the code.

In the examples provided, transistor counting has been performed to segregate transistors into their sub-components and then used to design this function such that each latch for the particular synchronizer is the same. This is deemed a reasonable approach as allowing all latches in a synchronizer design to be free to change though may yield a more optimal design is not likely to be a design which would be built in practice.

Table 1: Two flip-flop passgate synchronizer transistor numbering

Device Name	n-Device Id	p-Device Id
Inv buffer	1	2
Master flip-flop		
latch0	3,5,7,9,11	4,6,8,10,12
latch1	13,15,17,19,21	14,16,18,20,22
Slave flip-flop		
latch2	23,25,27,29,31	24,26,28,30,32
latch3	33,35,37,39,41	34,36,38,40,42

4.1.1 Two flip-flop passgate synchronizer example

To illustrate a concrete example, the two flip-flop passgate synchronizer found in:

`./ccm/Examples/Synchronizer Analysis/Two Flip-Flop Optimization/passgate design`

is used to show how the transistor numbering can be done and how it relates to creating the `populateADoptimizationVar.m` function.

The circuit definition used in the script which runs the optimization:

`TwoFlopPGSyncEKVOptimization.m`

is found in the MSPICE circuit definitions `./ccm/MSpice/coho/circuits` called `TWO_FLOP_PG_SYNC` and has a total of 42 transistors in its design. Lines 32 to 36 in the script define the transistor numbering scheme:

```
1 transistorNumbering = 1:1:numTx;
2 %        bufferInv     Master flip-flop            Slave
      flip-flop
3 %                  latch0         latch1           latch2
                 latch3
4 nDevices = [1,  3,5,7,9,11,   13,15,17,19,21,
      23,25,27,29,31,  33,35,37,39,41];
5 pDevices = [2,  4,6,8,10,12,  14,16,18,20,22,
      24,26,28,30,32,  34,36,38,40,42];
```

The above is broken down into Table 1 which explains how the *nDevice* and *pDevice* get accessed in the overall transistor width vector which defines the synchronizer circuit.

Each latch is the same as the one described in Section 3, and the Ids correspond to transistors within the latch definition. However for the purposes of distributing the variables properly in the overall width vector the information shown in Table 1 is sufficient.

With the transistor numbering information sorted, `populateADoptimizationVar.m` can be written. In this particular example it is done as follows:

```
1 function [adVar,numVar] = populateADoptimizationVar(
      voltageState,nDevices,pDevices)
```

```
2  %populateADoptimizationVar populates the AD variables for
       the computation
3  %of optimizing synchronizer circuits. This function is
       used to let the user
4  %decide what transistor widths are available to be
       decoupled for
5  %optimization purposes.
6  %   Inputs: voltageState  - the voltage state vector of
       the synchronizer
7  %           nDevices      - the vector of NMOS device
       widths in the design
8  %           pDevices      - the vector of PMOS device
       widths in the design
9  %   Outputs: adVAR is the AD variables which are used to
       compute the
10 %   gradient with respect to the desired transistor
       widths in the form of a
11 %   hessian object [voltageState;nDevices;pDevices] as a
       column vector
12
13 % This version of the function is to optimize a 2-flip
       flop passgate
14 % synchronizer where the coupling inverters are free to
       be individually
15 % sized.
16
17 % buffer = 1, passgate latch = 2 to 6
18 dev_indN = [1,2,3,4,5,6];
19 dev_indP = [1,2,3,4,5,6];
20
21 numVar = length(dev_indN) + length(dev_indP);
22
23 vCol = 1:length(voltageState);
24 nColBuf = length(voltageState)+1:length(voltageState)+1;
25 nCol    = length(voltageState)+2:length(voltageState)+
       length(dev_indN);
26
27 pColBuf = length(voltageState)+length(dev_indN) + 1:
       length(voltageState)+length(dev_indN) + 1;
28 pCol    = length(voltageState)+length(dev_indN) + 2:
       length(voltageState)+length(dev_indN) + length(
       dev_indP);
29
30 offsetV = length(voltageState);
31 bufOffset = 1;
32 offsetN = length(dev_indN)-1;
33 offsetP = length(dev_indP)-1;
34
35 AD_f = hessianinit([voltageState;nDevices(dev_indN)';
       pDevices(dev_indP)']);
```

```
36
37 syncState = diag(ones(1,length(voltageState)));
38
39 bufferN = 1;
40 latchN = diag(ones(1,length(dev_indN)-1));
41
42 bufferP = 1;
43 latchP = diag(ones(1,length(dev_indP)-1));
44
45 map = zeros(length(voltageState)+length(nDevices)+length(
       pDevices),length(voltageState)+length(dev_indN) +
       length(dev_indP));
46
47 %voltage State
48 map(1:offsetV,vCol) = syncState;
49 %n-devices
50 %buffer
51 map(1+offsetV:offsetV+bufOffset,nColBuf) = bufferN;
52 %latches
53 map(1+offsetV+bufOffset:offsetV+bufOffset+offsetN,nCol) =
       latchN;
54 map(1+offsetV+bufOffset+offsetN:offsetV+bufOffset+2*
       offsetN,nCol) = latchN;
55 map(1+offsetV+bufOffset+2*offsetN:offsetV+bufOffset+3*
       offsetN,nCol) = latchN;
56 map(1+offsetV+bufOffset+3*offsetN:offsetV+bufOffset+4*
       offsetN,nCol) = latchN;
57 %p-devices
58 map(1+offsetV+bufOffset+4*offsetN:offsetV+bufOffset+4*
       offsetN+bufOffset,pColBuf) = bufferP;
59 map(1+offsetV+bufOffset+4*offsetN+bufOffset:offsetV+
       bufOffset+4*offsetN+bufOffset+offsetP,pCol) = latchP;
60 map(1+offsetV+bufOffset+4*offsetN+bufOffset+offsetP:
       offsetV+bufOffset+4*offsetN+bufOffset+2*offsetP,pCol)
       = latchP;
61 map(1+offsetV+bufOffset+4*offsetN+bufOffset+2*offsetP:
       offsetV+bufOffset+4*offsetN+bufOffset+3*offsetP,pCol)
       = latchP;
62 map(1+offsetV+bufOffset+4*offsetN+bufOffset+3*offsetP:end
       ,pCol) = latchP;
63
64 adVar = map*AD_f;
65
66
67
68
69
70 end
```

The above code can be segregated into three parts: the `hessian` initialization,

the creation of the map to redistribute the variables in their appropriate location and finally the creation of the *adVar* which is used to compute $\frac{\partial G_{sync}}{\partial w}$ within the `nestedBisectionAlaysis` class.

Lines 18 to 35 define the parts of w which will be used as AD variables in the optimization, namely the buffer inverter and latch sub-circuit. We allow the input inverter to be independently sized to squeeze any performance out that circuit element which drives the input of the synchronizer.

Lines 37 to 62 create a mapping which is used to create the vector from the subset of w which fits the required form for the analysis.

Finally line 64 creates the vector used for the analysis.

4.2 `expandW.m`

This function unpacks the subset of variables used in `populateADoptimizationVar.m` back to its original size. Details on the function definition can be found in Section 6.4.15.

$$w = \begin{bmatrix} nDevices \\ pDevices \end{bmatrix} \tag{5}$$

In the two flip-flop passgate synchronizer the buffer inverter and passgate latch are the only AD variables which are distributed across the entire design. Thus to create the proper gradient at the end of the computation, the individual components of the resulting gradient vector needs to be expanded and is done as follows:

```
1  function resW = expandW(result)
2  %expandW takes the result passed into this function and
       expands the result
3  %to the appropriate size for the synchronizer in question
4  %   This expandW function is for a two flip-flop passgate
       synchronizer
5
6  bufN = 1;
7  bufP = 7;
8  latchN = 2:6;
9  latchP = 8:length(result);
10
11 resW = [result(bufN);
12           result(latchN);
13           result(latchN);
14           result(latchN);
15           result(latchN);
16           result(bufP);
17           result(latchP);
18           result(latchP);
19           result(latchP);
20           result(latchP)];
21
22 end
```

4.3 Optimization Constraints

The examples found in:

`./ccm/Examples/Synchronizer Analysis/Two Flip-Flop Optimization`

all apply constraints in a similar manner. We apply a barrier method to enforce no transistor be smaller than *minWidth* by first performing a change in variables.

$$\begin{aligned} u &= w/w_{min} - 1 \\ s &= u - 1/u \end{aligned} \tag{6}$$

With our new variable s, the gradient $\frac{\partial G_{sync}}{\partial w}$ is projected to the solution space of constant total width:

$$\begin{aligned} \frac{\partial G_{sync}}{\partial w}|_{fix} &= \frac{\partial G_{sync}}{\partial w} - r \cdot r \cdot \frac{\partial G_{sync}}{\partial w} \\ \frac{\partial w}{\partial s} &= w_{min}(1 + \frac{s}{\sqrt{s^2+4}}) \\ \frac{\partial G_{sync}}{\partial s} &= \frac{\partial G_{sync}}{\partial w}|_{fix} \cdot \frac{\partial w}{\partial s} \end{aligned} \tag{7}$$

where r is a vector of all ones normalized with the constant $1/\sqrt{|w|}$ ($|w|$ the length of the vector w).

However because our transformation is non linear, a small corrective term is applied after a gradient step to ensure that the total width remains constant using `fzero`. The new transistor widths are then used to determine if we have converged or continue the optimization.

The jamb and robust jamb design require additional constraints which are applied in the function `pullDownSizing.m` (found in their respective folders) which add margin to the design for robustness to Process, Voltage and Temperature (PVT) variation.

5 MSPICE Circuit

This section is intended to be a reference to all features available in the `circuit.m` class and describe the available subclasses of `circuit`. Included within is a comprehensive list of methods that are available to the user. The private methods of the classes will be denoted as such so that a user looking to extend functionality is aware of their existence and will be clearly marked when being described.

The `circuit.m` class in MSPICE is the class which defines circuits, creates connectivity matrices, a circuit path object and provides methods to query information about the circuit to extract specific information relevant to the defined circuit. It has as sub-classes: `coho_leaf.m` and `coho_vsrc.m`.

A list of defined components and circuits is found in the MSPICE library list Section 8

5.1 Circuit `TEMPLATE.m`

Found in `./ccm/libs/coho/circuits/TEMPLATE.m` is a blank template file that can be used to define user circuit definitions. If these are deemed to be components that make up larger circuits they can be saved in `./ccm/libs/coho/components`. Such examples of components are `INV4.m`, `NAND2.m`, ect. A full list of available

components as v2.0 of this manual can be found in **??**. The template can be decomposed in the following:

- The template is of class `circuit`
- The properties by default are *get* 'public' and *set* 'private'. These properties are generally inputs and outputs of the circuit.
- The constructor for the circuit: `TEMPATE(name,wid,rlen)` requires a circuit *name*, *wid* (for transistor widths), and *rlen* an optional parameter that can modify the relative length of the transistor sizes (legacy from version 1 of coho - generally not provided or set to 1).
- Following the constructor definition, is code to provide checks and throw an error if the inputs are incorrect, along with the ability to automatically set default values.
- This is then followed by constructing the superclass `circuit` object which gets assigned the variable *this* which is used to access methods of the `circuit` class and any methods unique to the current subclass.
- Finally the body of the circuit definition is filled in and then must be finalized by calling `this.finalize` at the end of the circuit definition.

5.2 `circuit.m` class

This section describes all the methods available in the `circuit.m` class. The methods denoted private/protected are labelled as such. A brief description on what each method does is written here as well as the required inputs. I feel like the code is reasonably well documented with comments, however there are bound to be places where the comments are wrong thus the descriptions on what is supposed to happen is kept up to date here.

The `circuit.m` class is found in:

```
./ccm/MSpice/circuit.m
```

A `circuit` has the following attributes once *finalized* (see method `finalize`):

1. *`name`*: The name of the circuit
2. *`nodeNum`*: The number of `nodes` in the circuit
3. *`ports`*: The visible ports at the top level of the circuit (can be multiple hidden/internal ones)
4. *`cctPath`*: A MATLAB `struct` variable which map node names to node numbers i.e. `myCircuit.cctPath.q` = 5
5. *`finalized`*: a boolean variable which indicates if the `circuit` has been *finalized*.
6. `flattened`: a boolean variable which indicates if the `circuit` has been *flattened* (required for vectorization)

7. *vectorized*: a boolean variable which indicates if the `circuit` has been *vectorized*
8. *elements*: list of `circuit` elements
9. *maps*: list of `circuit` maps (includes sub-circuits)
10. *conns*: list of connections - used to create *maps*
11. *grp_cids*: list of grouped `circuit element` ids
12. *grp_maps*: list of merged maps of the same type of `circuit elements`.

In the remainder of this section, the methods available will be described in their own sub-sections.

5.2.1 `circuit` constructor (protected)

The constructor for a `circuit` object is *protected* and called as follows:

```
myCircuit = circuit('name')
```

The above returns a `circuit` object with the *name* attribute set to the input string *name*.

5.2.2 `add_port` (protected)

This function adds a port to the circuit if it is a new port and called as follows:

```
port = myCircuit.add_port(port)
```

where *port* is the `port` to be added to the circuit.

5.2.3 `add_element` (protected)

This function adds an element to the circuit if it is new and called as follows:

```
element = myCircuit.add_element(element)
```

where *element* is the element to be added if it is new.

5.2.4 `connect` (protected)

This function connects ports together and is called as follows:

```
myCircuit.connect(port1,port2,varargin)
```

where *port1*, *port2* and *varagin* are the ports to connect together. At minimum `connect` requires two ports to connect together.

5.2.5 `finalize` (protected)

This function *finalizes* the `circuit` object and must called before it can be used in a `testbench`:

```
myCircuit.finalize
```

`finalize` takes no arguments, but initializes any outstanding `circuit` attributes.

5.2.6 `ifc_simu`

This function checks if the `circuit` object can be simulated:

```
response = myCircuit.ifc_simu
```

where `response` is a boolean variable.

5.2.7 `is_vsrc`

This function checks if it is a voltage source and takes no arguments:

```
response = myCircuit.is_vsrc
```

where `response` is a boolean variable.

5.2.8 `getCircuitMap`

This function returns a cell array with the `circuit element` properties and their associated `circuit maps`:

```
circuitMapping = myCircuit.getCircuitMap
```

where `circuitMapping` is the cell array result. Used for obtaining `circuit element` and map information at simulation time.

5.2.9 `getDevices`

This function returns a set of `devices` which correspond to the indices *`query`* of the `circuit`.

```
devices = myCircuit.getDevices(query)
```

5.2.10 `getDevicesConnectedToNode`

This function returns a set of `deviceMaps` and associated device `names` which have any node connected the query *`node`*.

```
[deviceMaps,names] = myCircuit.getDevicesConnectedToNode(node)
```

5.2.11 `getDevicesByIndex`

This function returns the `device` associated with the input *`index`* of the `circuit`, used as part of `getDevices`.

```
device = myCircuit.getDevicesByIndex(index)
```

5.2.12 `getElementWiseParameters` (private)

This function returns a list of device parameters `listParams` for the set of input *`circuitElements`*.

```
listParams = myCircuit.getElementWiseParameters(circuitElements)
```

5.2.13 `vectorize`

This function *vectorizes* the `circuit` for faster simulation by grouping like `circuit` elements together.

```
myCircuit.vectorize
```

5.2.14 `flatten`

This function *flattens* the `circuit` hierarchy. Will undo `vectorize` if the `circuit` is previously `vectorized`.

```
myCircuit.flatten
```

5.2.15 `is_leaf`

This function returns a boolean `result`. If the `circuit` is a `coho_leaf`, then the result is `true` otherwise it is `false`.

```
result = myCircuit.is_leaf
```

5.2.16 `elemNum`

This function returns the number of elements at the top level `circuit` definition. To get the number of all elements in the `circuit`, `flatten` the `circuit` first.

```
numElement = myCircuit.elemNum
```

5.2.17 `find_port_index`

This function returns the `index` of the queried *`port`*.

```
index = myCircuit.find_port_index(ports)
```

5.2.18 find_node_name

This function returns the names of the queried *index*.

```
names = myCircuit.find_node_name(index)
```

5.2.19 find_node_name_short

This function returns the names in short form of the queried *index*.

```
names = myCircuit.find_node_name_short(index)
```

5.2.20 find_path

This function returns as a struct the path to the queried *index*.

```
path = myCircuit.find_path(index)
```

5.2.21 print_circuit_tree

This function prints out the entire circuit hierarchy tree as a list of strings. Optional to this function is a user defined *prefix* which is added to the beginning of each string.

```
strings = myCircuit.print_circuit_tree(prefix)
```

5.2.22 print_status

This function prints out the status of the circuit with the option of adding a *prefix* to the beginning of the string.

```
strings = myCircuit.print_status(prefix)
```

5.2.23 print_nodes

This function prints out the *nodes* of the circuit as a set of strings.

```
strings = myCircuit.print_nodes
```

5.2.24 create_path

This function creates a struct pathNodes which has as <'key',value> pairs the node name and associated index. Created upon the construction of the circuit. Useful for auto-completion search of nodes within the circuit.

```
pathNodes = myCircuit.create_path
```

5.2.25 add_path

This function adds a pathNodes object to an already existing *pathNodes* object with the given *nodeId* and *path* and *name*. Used to in create_path to recursively traverse the circuit tree.

```
pathNodes = myCircuit.add_path(nodeId,path,pathNodes,name)
```

5.3 coho_leaf.m class

The coho_leaf.m class is a subclass of circuit.m and describes MOSFET components, resistors, capacitors and inductors. These coho_leaf nodes do not contain sub-circuit definitions. A coho_leaf subclass of circuit has the following attributes:

1. *subinfo*: a struct which contains:
   (a) *device*: the device name
   (b) *I_factor*: a column vector which assigns sign for the current into the nodes of the device
   (c) *C_factor*: a column vector which assigns sign for the capacitance of the nodes of the device
2. *params*: a struct with the required fields for the device

A coho_leaf is constructed as follows:

```
myLeaf = coho_leaf(subinfo,'name',params)
```

where *subinfo* and *params* are directly set to the coho_leaf attributes, and *'name'* is the name used in the constructor of the circuit superclass.

5.3.1 resistor

A resistor element is constructed as follows:

```
myR = resistor('name',r,ctg)
```

Where *'name'* is the name of the resistor element, *r* is the resistance in [Ω] value and *ctg* is the connect-to-ground flag.

A resistor has two ports x and y and the current flowing into node x: $I_x = \frac{V_y - V_x}{r}$ and similarly the current flowing into node y: $I_y = \frac{V_x - V_y}{r}$.

5.3.2 capacitor

A capacitor element is constructed as follows:

```
myC = capacitor('name',c,ctg)
```

Where *'name'* is the name of the capacitor element, *c* is the capacitance in $[F]$ value and *ctg* is the connect-to-ground flag.

A capacitor has two ports x and y. The current in a capacitor is determined by solving $I_c = \frac{dV}{dt}C$. During simulation we integrate $\frac{dV}{dt}$ to find $V(t)$, thus I_c at integration time is determined by solving for $I_c + I_x = 0$ and vice versa for I_y where I_x and I_y are the sum of the current contributions from the other circuit elements.

5.3.3 inductor

A inductor element is constructed as follows:

```
myL = inductor('name',l)
```

as of the writting of v2.0 of this manual, this is not yet implemented.

5.3.4 pmos

A pmos element is constructed as follows:

```
myPmos = pmos(name,varargin)
```

where *name* is the name of the pmos device. The constructor parameter *varargin* supports three different instantiation methods. The preferred approach in v2.0 and for future additions is by passing in either a struct object or by the MATLAB convention of <'key',value> pairs, e.g.:

1. myPmos = pmos('p',1,1,0): this version is discouraged but has the following signature:
   pmos(*name*,*width*,*ctg*,*rlen*)
   (a) *name*: device name
   (b) *width*: device width in $[m]$
   (c) *ctg*: connect-to-ground flag for source node
   (d) *rlen*: relative length of device (usually set to 1)
2. myPmos = pmos('p','wid',1,'rlen',1): where 'p' is the device name, followed by the <'key',value> pairs
3. myPmos = pmos('p',params), where 'p' is the device name and params is a struct which contains the appropriate fields and values for the pmos device

A pmos device requires the following fields:

1. *wid*: transistor width
2. *rlen*: relative length of the transistor

An optional field *'model'* is available to specify a particular model to use in the evaluation at simulation time, otherwise the tbOptions (see Section 6.1.1) default model is used.

A pmos devices has 4 ports:

1. *drain*: myPmos.d
2. *gate*: myPmos.g
3. *source*: myPmos.s
4. *body*: myPmos.b

5.3.5 `nmos`

A `nmos` element is constructed in the same way as a `pmos` (see above Section 5.3.4), but replace `pmos` with `nmos`.

5.4 `coho_vsrc.m` class

The `coho_vsrc.m` class is a subclass of `circuit.m` and describes voltage sources for MSPICE and has `ports` p and m for the plus and minus ports of the `source`. A source is constructed as follows:

```
mySource = coho_vsrc(name,ctg)
```

where *`name`* is the name of the source and *`ctg`* is the connect-to-ground flag.

Most voltage sources are connected to ground however sources can be included in the middle of `circuit` definitions where their `ports` p and m are connected to nodes of interest.

The voltage sources described here return their voltage for the time point(s) as `mySource.V(t)` unless otherwise specified.

5.4.1 `vsrc`

A `vsrc` constant voltage source is constructed as:

```
vdd = vsrc('name',voltage,ctg)
```

where *`'name'`* is the name of the source, *`voltage`* is the value the voltage source outputs and *`ctg`* is the connect-to-ground flag (generally) set to true.

To evaluate the voltage source:

```
V = vdd.V(t)
```

will return a row vector of *`voltage`* the same length as the input `t`.

5.4.2 `vpulse`

A `vpulse` source voltage is constructed as:

```
myPulse = vpulse('name',range,period,d,ctg)
```

1. *`name`*: the name of the source
2. *`range`*: the range of the voltage pulse in the form $[v_{lo}, v_{hi}]$
3. *`period`*: the period of the pulse in the from $[t_{rise}, t_{high}, t_{fall}, t_{low}]$
4. *`delay`*: the delay before the pulse starts t_{delay}
   (optional - if not provided it is set to 0)
5. *`ctg`*: the connect-to-ground flag
   (optional - it not provided default is `true`)

This source produces periodic pulses by taking `mod`$(t - t_{delay}$,$t_P)$ where $t_P = t_{rise} + t_{high} + t_{fall} + t_{low}$ as the time points t to evaluate this source.

Mathematically this source is described as:

$$v(t) \quad = \quad \begin{cases} v_{lo} & t \in [0, t_{delay}] \\ v_{lo} + \frac{v_{hi}-v_{lo}}{t_{rise}} t & t \in (t_{delay}, t_{rise}] \\ v_{hi} & t \in (t_{rise}, t_{high}] \\ v_{hi} - \frac{v_{hi}-v_{lo}}{t_{fall}} t & t \in (t_{high}, t_{fall}] \\ v_{lo} & t \in (t_{fall}, t_{low}] \end{cases} \tag{8}$$

5.4.3 `vsrcLinear`

A `vsrcLinear` source is constructed as:

```
myLinearSrc = vsrcLinear('name',slope,ctg)
```

where *'name'* is the name of the source, *slope* is the slope m of the source and *ctg* is the connect-to-ground flag.

Mathematically this source is described as:

$$v(t) \quad = \quad mt \tag{9}$$

5.4.4 `vsin`

A `vsin` source is constructed as:

```
myVsin = vsin('name',angularFrequency,phi,offset,amplitude,ctg)
```

1. *'name'*: is the source's name
2. *angularFrequency*: is the angular frequency w of the source
3. *phi*: is the phase offset ϕ of the source
4. *offset*: is the offset b of the source
5. *amplitude*: is the amplitude A of the source
6. *ctg*: is the connect-to-ground flag

Mathematically this source is described as:

$$v(t) \quad = \quad A\sin(wt + \phi) + b \tag{10}$$

5.4.5 `vtanh`

A `vtanh` source is constructed as:

```
myVtanh = vtanh('name',amplitude,offset,gain,delay,ctg)
```

1. *'name'*: is the source's name

2. *amplitude*: is the amplitude A of the source
3. *offset*: is the offset b of the source
4. *gain*: is the gain g of the source and controls how quickly the source transitions from the *low* state to the *high* state
5. *delay*: is the delay t_{delay} of the source which time shifts when the transition occurs
6. *ctg*: is the connect-to-ground flag

Mathematically this source is described as:

$$v(t) = A\tanh\left(g(t - t_{delay})\right) + b \tag{11}$$

5.4.6 vtanhClock

A `vtanhClock` source is constructed as:

```
clk = vtanhClock(name, angularFrequency, offset, amplitude, gain, delay, ···
            startHigh, ctg)
```

The output of this source is mathematically described as:

$$v_{source}(t) = \mathcal{H}(t - t_{delay})\left(A\tanh\left(g\sin\left(\omega(t - t_{delay}) + \phi\right)\right) + b\right) \tag{12}$$

Where $\mathcal{H}$ is the Heaviside function:

$$\mathcal{H}(t) = \begin{cases} 1, t > 0 \\ 0, t < 0 \end{cases} \tag{13}$$

1. *name*: is the voltage source's *name*
2. *angularFrequency*: is the angular frequency ω of the oscillator
3. *offset*: is the offset of the oscillator b
4. *amplitude*: is the amplitude A of the oscillator
5. *gain*: is the gain g of the oscillator, this parameter tunes how quickly the oscillator swings from the *low* to the *high* state (or vice versa)
6. *delay*: is the time delay t_{delay} before the oscillator starts oscillating
7. *startHigh*: is a boolean flag which allows the clock to be held in its high state before it begins to oscillate
8. *ctg*: is the connect-to-ground flag (often set to true)

5.4.7 `vtanhDelay`

A `vtanhDelay` is constructed as:

`din = vtanhDelay(`*`name`*`,`*`amplitude`*`,`*`offset`*`,`*`gain`*`,`*`ctg`*`)`

The output of this source takes in two parameters t and t_{in} and is mathematically described as:

$$v(t, t_{in}) \quad = \quad A \tanh\left(g(t - t_{in})\right) + b \tag{14}$$

1. *`name`*: is the source's name
2. *`amplitude`*: is the amplitude A
3. *`offset`*: is the offset b
4. *`gain`*: is the gain g which controls how fast the signal changes from it's low state to high state (or vice versa)
5. *`ctg`*: is the connect-to-ground flag (often set to true)

To evaluate this source, `V = din.V(t,delay)`, i.e. the user specifies the delay at evaluation time to determine the delay of the source.

6 MSPICE Testbench

Found in this section of the manual are the subclasses of the `testbench.m` class and their associated methods. Included in this section is a description of the required user defined functions which are necessary for some of the methods.

6.1 `testbench.m` class

A `testbench` is create with the following signature:

`myTestbench = testbench(`*`circuit`*`,`*`ports`*`,`*`sources`*`,`*`model`*`,`*`process`*`,`*`options`*`)`

1. A *`circuit`* definition such as the one described in Section 2.1.2.
2. The *`circuit`*'s *`ports`* are an ordered cell array, i.e. {`myOsc.vdd,myOsc.gnd`}
3. The *`sources`* are external sources that connect to the *`circuit`*'s *`ports`* in the order they appear as a cell array, i.e. {`Vdd,Gnd`}
4. The *`model`* is the default `model` that the `testbench` uses for `simulation` (optional - if not set, default `model` is set to *interp*)
5. The *`process`* is the `process` for the `testbench`'s *`model`* (optional - if not set, default `process` is set to *PTM 180nm*)
6. The *`options`* is the `option` object that determine parameters for the `testbench` at simulation time
   (optional - if not set see *default options*)

A `testbench` has the following attributes:

1. *`circuit`*: the `testbench circuit`
2. *`sources`*: the list of sources for the `circuit`
3. *`map`*: the mapping between `circuit` ports and voltage sources
4. *`modelDictionary`*: the `modelDictionary` object which dispatches the `model` to be used at simulation time
5. *`defaultModel`*: the default `model` to be used at simulation time
6. *`tbOptions`*: MATLAB `struct` with a list of `testbench` options, can include nested `structs`
7. *`odeSave`*: a cell array of `ode` data if the `testbench` is simulated in *debug* mode.
8. *`simVoltage`*: the voltage vector saved after simulation, used to display data.
9. *`simTime`*: the associated time points with the *`simVoltage`* data.
10. *`circuitMap`*: the `circuitMapping` data for the `testbench circuit`. (protected and private)
11. *`matrixMap`*: the `circuit` map that distributes voltages to the appropriate nodes of the `circuit` devices. (protected and private)

The default `testbench` options `tbOptions` which are used if no options are specified in the constructor are as follows:

1. *`'capModel'`* - `'gnd'`
2. *`'capScale'`* - 1
3. *`'vdd'`* - 1.8
4. *`'temp'`* - 298
5. *`'numParallelCCTs'`* - 1
6. *`'debug'`* - `false`

The list of available `testbench options` are outlined in the following section.

6.1.1 Testbench options: `tbOptions`

The `testbench options` follow MATLAB's `<'string',value>` pair convention. The `tbOptions struct` is a means of adding parameters and options to the `testbench` which are specific to the `model` being used. In this way this object becomes a means in which necessary information can be passed to the available models.

Listed here are the *keys* used in the `testbench` class. Additional `structs` and options are available for the other subclasses of the `testbench` class.

1. *'capModel'*:
   - *'gnd'*: all capacitors are referenced to ground
   - *'full'*: includes inter-node capacitance if the model supports it and models secondary effects such as Miller capacitance for `MOS devices`
2. *'capScale'*: scales the numerical integration by the specified value. Can help with numerical stability of integration, scaling factor is reversed after simulation
3. *'vdd'*: global V_{dd} level for the `circuit` where this information is needed.
4. *'temp'*: Temperature in degrees Kelvin, if the model supports temperature dependence then this indicates the temperature the `circuit` is to be simulated at.
5. *'numParallelCCTs'*: number of simultaneous simulations of the `circuit` is performed at one time. Necessary in the `nestedBisection` class.
6. *debug*: Boolean variable which if `true` saves in the `testbench` attribute as a cell array `odeSave`:
   (a) `odeSave{1}`: the number of data points saved
   (b) `odeSave{i}`: the ith data point saved as:
      i. *data{1}*: C the capacitance matrix/column vector of the `circuit`
      ii. *data{2}*: I the column current vector of the `circuit`
      iii. *data{3}*: dV the derivative vector of the `circuit`
      iv. *data{4}*: t the time point

6.1.2 `simulate`

This function simulates the `circuit` within the `testbench` over a user specified time span with the option to specify the initial conditions and specific integrator options. By default the integrator used is MATLAB's `ode45` integrator with default setting.

`myTestbench.simulate(`*timeSpan*`,`*v0*`,`*options*`)`

1. *timeSpan*: The integration time span in the form $[t_{start}, t_{stop}]$
2. *v0*: a user supplied initial voltage vector which also includes initial voltages for the `circuit` sources.
   (optional - if not supplied *v0* is a vector of all zeros)
3. *options*: The user supplied integration options, if there is a field *'Integrator'* then the integrator is set to the user supplied integrator (see MATLAB's integrators `https://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html`)
   (optional - if not supplied the integrator is MATLAB's `ode45` with default settings)

6.1.3 dV_ode

This function is the derivative function for the numerical integration method `simulate`. Can be called independently of the `simulate` method to get the derivative of the circuit at a particular time t and voltage state V.

`myTestbench.dV_ode(`*t*`,`*v_ode*`)`

1. *t*: the specified time point for the derivative of the `circuit`
2. *v_ode*: the specified voltage state of the `circuit`

6.1.4 packageTestbenchOption

This function packages `testbench options` if supplied in the constructor as a list of `<'string',value>` pairs instead of a `struct`.

`options = myTestbench.packageTestbenchOptions(`*tbOptions*`)`

where *tbOptions* is a cell array formatted with {{'string'},{value}$\cdots$} pairs.

6.1.5 getTestbenchOptions

This function returns the `tbOption` attribute of the `testbench`

`tbOptions = myTestbench.getTestbenchOptions`

6.1.6 setTestbenchOptions

This function sets the `tbOption` attribute of the `testbench` to the user supplied `tbOptions`.

`myTestbench.setTestbenchOptions(`*tbOptions*`)`

6.1.7 setSimTimeVoltage

This function allows the user to set their own (potentially previously saved) set of `simVoltage` and `simTime` variables to the associated `testbench` attributes.

`myTestbench.setSimTimeVoltage(`*time*`,`*voltage*`)`

where *time* and *voltage* are the user specified arrays of data.

6.1.8 dim

This function returns the dimension of the `circuit`.

`n = myTestbench.dim`

where `n` is the dimension of the `circuit`

6.1.9 `name`

This function returns the `circuit` attribute *name*.

```
name = myTestbench.name
```

6.1.10 `dut`

This function returns the `circuit` of the `testbench`.

```
circuit = myTestbench.dut
```

6.1.11 `inputNum`

This function returns the number of input sources to the `testbench circuit` .

```
n = myTestbench.inputNum
```

6.1.12 `inputs`

This function returns all the sources or the specified source(s) by the user.

```
sources = myTestbench.inputs(indices)
```

where *indices* are the indices of the `sources` requested.

6.1.13 `inputNodes`

This function returns the `map` attribute of the `testbench` which maps indices to sources.

```
map = myTestbench.inputNodes
```

6.1.14 `find_port_index`

This function returns the index for the given `ports` to find.

```
indices = myTestbench.find_port_index(ports)
```

6.1.15 `getIdsDevices`

This function returns the current (flowing from drain to source in the context of a `MOS device`) at the specified time point(s) and state voltage(s).

```
currents = myTestbench.getIdsDevices(query,time,voltage)
```

1. *query* is the user specified device query, requires knowledge of the node numbers of the device
2. *time* the time point(s) of interest
   (optional - if not supplied the `simTime` from the `testbench` is used)

3. *voltage* the voltage(s) state of the `circuit`
   (optional - if not supplied the `simVoltage` of the `testbench` is used)

This function is used in the `explainNodes` method.

6.1.16 explainNodes

This function returns a plot for each `node(s)` requested by the user. Each plot shows the individual current contributions going into the node along with its voltage and any other supplied voltages supplied by the user and a second plot which shows the cumulative sum of the current contributions.

`myTestbench.explainNodes(`*nodes*`,`*time*`,`*voltage*`)`

1. *nodes*: The `nodes` of interest to be explained
2. *time*: The specified time point(s) of interest to conside.
   (optional - if not supplied the `testbench simTime` attribute is used)
3. *voltages*: The specified voltage state(s) to be used.
   (optional - if not supplied the `testbench simVoltage` attribute is used)

6.1.17 setOdeSave

This function sets the `testbench` attribute `odeSave` object, used to update the object at simulation time if in `debug` mode.

`myTestbench.setOdeSave(`*ind*`,`*data*`)`

where *ind* is the number of data points in *data* which contains the diagnostic information for a particular simulation.

6.1.18 getOdeSave

This function retrieves the `testbench` attribute `odeSave`.

```
odeSave = myTestbench.getOdeSave
```

6.1.19 eval_vs

This function is used to evaluate the voltage sources of the `circuit` at the specified *time* point during simulation.

`voltage = myTestbench.eval_vs(`*time*`)`

6.1.20 is_src

This is a function which returns a logical vector *map* which allows the `simulation` method to separate voltage sources and `circuit` state variables.

```
map = myTestbench.is_src
```

6.1.21 computeMapMatrix

This function takes the testbench circuit and creates a connectivity matrix N which makes up the protected and private attribute matrixMap of the testbench .

```
map = myTestbench.computeMapMatrix
```

6.1.22 vfull (private)

This function expands the circuit voltage state vector to include the voltage source voltages of the circuit . Used at simulation time to distribute voltages to the various circuit devices.

```
voltage = myTestbench.vfull(t,v_ode)
```

where *t* is the simulation time point and *v_ode* is the simulated circuit voltage state.

6.2 nestedBisection.m class

This section covers the functions found in the nestedBisection.m class and any functions which override the testbench class. The order in which the functions are described is not necessarily in the same order as they appear in the code.

The nestedBisection class has the following attributes:

1. *inputSource*: the input source parametrized with by t_{in} which changes when the d_{in} transition occurs
2. *dinInterval*: the nestedBisection current time interval where $t_{in} \in [t_{hi}, t_{lo}]$ would cause the synchronizers to exhibit metastable behaviour to at least time t
3. *inputSourceIndex*: the index which corresponds to the input source among the list of sources for the synchronizers
4. *clockSource*: the clock source for the synchronizers
5. *nodeMask*: the nodes of interest for the nested bisection algorithm to investigate
6. *synchronizerCCTMap*: synchronizer circuit device mapping information/vectorized attributes for the circuit definition (protected)
7. *syncMatrixMap*: a sparse matrix mapping for all numParallelCCTs being simulated (protected)

A nestedBisection testbench is created using the following signature:

```
myNBTestbench = nestedBisection(circuit,ports,sources,inputSource,···
                inputInterval,clockSource,mask,model,process,options)
```

1. *circuit*: is a synchronizer circuit which has an input d_{in} which is parametrized by the value t_{in}.
2. *ports*: are the synchronizer's *ports* excluding d_{in} input as a cell array in the order that they are listed, e.g. {`mySync.vdd,mySync.gnd,mySync.clk,mySync.reset`}.
3. *sources*: are the sources input as a cell array connecting to the synchronizer's *ports* in the order in which they appear, e.g. {`Vdd,Gnd,Clk,Reset`}.
4. *inputSource*: is the `circuit`'s input source index and `voltage source` for the synchronizer's d_{in} signal, passed as a cell array. e.g. {`sync.d,din`}
5. *inputInterval*: is the initial interval $[t_{hi}, t_{lo}]$ on which the nested bisection algorithm makes its search towards finding t_{in} which causes the synchronizer to exhibit metastable behaviour out to time t.
6. *clockSource*: is the clock source for the synchronizer, used to determine when the clock edges occur to switch which node in *mask* to consider.
7. *mask*: is an array of nodes which correspond to which clock edge is considered to determine the last trajectory to settle high and the last one to settle low.
8. *model*: is the default `model` that the `testbench` uses for `simulation`. (optional - if not supplied the default `model` is the *MVS* `model`)
9. *process*: is the `process` for the `testbench`'s *model*. (optional - if not supplied the default is the *PTM 45nmHP* `process`)
10. *options*: is the `option` object that determine parameters for the `testbench` at simulation time. (optional see default `tbOptions` when not supplied below).

The default `tbOptions` in constructing a `nestedBisection testbench` are as follows:

1. *'capModel'* - 'full'
2. *'capScale'* - 1e10
3. *'vdd'* - 1.0
4. *'temp'* - 298
5. *'numParallelCCTs'* - 10
6. *'stepOut'* - 1
7. *'polyFitDegree'* - 3
8. *'polyFitError'* - 1e-1
9. *isAD* - `false`
10. *transSettle* - 1e-10
11. *'plotOptions'* - `false`

12. *'integratorOptions'* - integratorOptions: `odeset('RelTol',1e-6,'AbsTol',1e-6)`
13. *'clockEdgeSettings'* - clkEdgeSettings:
    (a) *'low'* - 0.48
    (b) *'high'* - 0.52
    (c) *'clkEdgeSpread'* - 20e-12
    (d) *'numberOfEdges'* - 3
14. *'digitalSimOptions'* - digitalSimOptions:
    (a) *'vdd'* - 1.0
    (b) *'minSimTime'* - 6e-10
    (c) *'threshold'* - 0.1
    (d) *'stopGain'* - 30
15. *'integrator'* - `@ode45`
16. *'debug'* - `false`

6.2.1 nested bisection options: `tbOptions`

Listed here are the `testbench` options which are not listed in Section 6.1.1, those listed in the `testbench` also apply here. The `struct` follows the `<'string',value>` pair as described in the `testbench options`.

1. *'stepOut'* - The number of simultaneous trajectories to step out for robustness of the bisection algorithm
2. *'polyFitDegree'* - The degree of the polynomial fit when testing for when the linearity assumption fails
3. *'polyFitError'* - The amount the fit polynomial can diverge from the remaining trajectories in a Frobenius norm sense, i.e. $||\mathbf{V}_{states} - \mathbf{V}_{fit}||_2$ where $\mathbf{V}_{states} = \begin{bmatrix} V_1 & V_2 & \cdots & V_{numParallelCCTs} \end{bmatrix}$ and $\mathbf{V}_{fit} = \begin{bmatrix} V_{fit} & V_{fit} & \cdots & V_{fit} \end{bmatrix}$ such that V_{fit} is the fit polynomial of degree *'polyFitDegree'* to the set of trajectories $V_1, V_2, \cdots, V_{numParallelCCTs}$ at time t.
4. *isAD* -
    (a) `true`, it sets the voltage state of the synchronizer to an AD variable - this would compute the Jacobian for all *numParallelCCTs* at each time point of simulation.
    (b) `false` simulate without AD variables (much faster simulation time with this option)
5. *transSettle*: the allotted time where initial transients due to initial conditions are allowed to settle. Data before *transSettle* is not saved.
6. *'plotOptions'*: object for future consideration that can be used in the `bisectionPlotV` and `bisectionPlotBeta` methods.

7. *'integratorOptions'*: object set by MATLAB `odeset` see `https://www.mathworks.com/help/matlab/ref/odeset.html`

8. *'clockEdgeSettings'*: is a `struct` that contains settings used in the `findClockEdges` method with the following settings:

   (a) *'low'*: the lower bound on a clock edge transition.

   (b) *'high'*: the upper bound on a clock edge transition.

   (c) *'clkEdgeSpread'*: The difference in time between one *'low'* and *'high'* time interval before a new clock edge is deemed detected

   (d) *'numberOfEdges'*: The number of clock edges to find

9. *'digitalSimOptions'*: is a `struct` that contains settings that are used in the wrapper method `stopCriteria` which calls the user supplied `stopToDigitalSim.m` function (see Section 6.2.4). The defaults are:

   (a) *'vdd'*: global supply voltage for the `testbench` (assumed that this is the level which is considered *high* in the simulation)

   (b) *'minSimTime'*: minimum simulation time to be elapsed before termination is considered

   (c) *'threshold'*: threshold that all signals within the synchronizer need to attain before termination

   (d) *'stopGain'*: scaling factor which changes the speed of the stopping function's transition from negative to positive

10. *'integrator'*: integrator to be used in the simulation

6.2.2 `simulate`

This function overrides the `testbench` simulate function and launches the nested bisection algorithm for the synchronizer circuit in the `nestedBisection testbench`. The call has the following signature:

```
bisectionRuns = myNBTestbench.simulate(filename,timeSpan,tCrit,v0)
```

1. `bisectionRuns`: is a cell array which contains the nested bisection simulation results. Each index i into the cell array (starting from position 2) corresponds to a bisection epoch which includes:

   (a) `bisectionRuns{1,i}`: the times for t_{hi} and t_{lo} as $[t_{hi}, t_{lo}]$

   (b) `bisectionRuns{2,i}`: the cumulative fraction that the initial time interval has shrunk

   (c) `bisectionRuns{3,i}`: the *startTime* and *endTime* of each bisection simulation as $[t_{start}, t_{end}]$.

   (d) `bisectionRuns{4,i}`: The voltage state of the synchronizer that settled high and the voltage state of the synchronizer that settled low at t_{eola_K}, the end of the linear time of the Kth epoch of the nested bisection algorithm as $[V_H, V_L]$

(e) `bisectionRuns{5,i}`: the epoch number

(f) `bisectionRuns{6,i}`: the numerical approximation of $\beta(t) \approx \frac{V_H(t)-V_L(t)}{\Delta t_{in}}$

(g) `bisectionRuns{7,i}`: the indices of the synchronizer circuit which settled high and the one which settled low as $[ind_L, ind_H]$

(h) `bisectionRuns{8,i}`: the trajectory of the synchronizer that settled high at the ith simulation $V_H(t)$

(i) `bisectionRuns{9,i}`: the trajectory of the synchronizer that settled high at the ith simulation $V_L(t)$

(j) `bisectionRuns{10,i}`: the time point $t \in [t_{start}, t_{end}]$

(k) `bisectionRuns{11,i}`: the quantity Δt_{in}

(l) `bisectionRuns{12,i}`:the time points of the nested bisection epoch, i.e. the time points where the synchronizer can be analysed using a linear model saved as $t \in [t_{start}, t_{eola_K}]$

(m) `bisectionRuns{13,i}`: numerical integration debug information as described in Section 6.1.1 if `debug = true`

(n) `bisectionRuns{14,i}`: all simultaneously simulated synchronizer voltage states

2. *`filename`*: a string which saves the nested bisection `bisectionRuns` simulation data to `'filename.mat'`

3. *`timeSpan`*: the overall time span the numerical integration is to take place.

4. *`tCrit`*: the time by which the synchronizer needs to just meet the user defined `threshold` for digital levels. Also used as a check to determine if the `nestedBisectionStop.m` (see Section 6.2.5) function is to be called to terminate the algorithm.

5. *`v0`*: initial voltage state to be applied to the synchronizers.
(optional - if not supplied *v0* is a vector of all zeros)

The simulate function makes calls to `stopToDigitalSim` (indirectly via `stopCriteria`) and `nestedBisectionStop` which give the user the flexibility to terminate a nested bisection epoch and the entire nested bisection algorithm. These functions must be defined and supplied by the user for `simulate` to work properly.

The `nestedBisectionStop` function is only called after the simulation time t has time points that are greater than *`tCrit`*. The algorithm terminates if `nestedBisectionStop` returns `true`, otherwise the algorithm continues. If the `nestedBiesctionStop` never returns `true`, then the algorithm runs forever. – A useful note to observe when debugging.

6.2.3 `stopCriteria` (protected)

This function is a wrapper to the user provided `stopToDigitalSim.m` function and used to end a nested bisection epoch.

```
[value,isterminal,direction] = myNBTestbench.stopCriteria(t,V)
```

Where `value`,`isterminal` and `direction` are values for MATLAB's `ode event` function. Internally this function computes $\dot{V}$ and the full voltage state of the synchronizer by calling `vfull` with the inputs *t* and *V*.

6.2.4 stopToDigitalSim.m

During the `nestedBisection simulate` method there is a call to the `stopCriteria` which is a wrapper to the user defined `stopToDigitalSim.m` function.

The nested bisection algorithm is greatly accelerated when a stopping condition can be determined for which the digital state of the synchronizer is no longer changing. Finding a criteria to determine this and terminating the nested bisection algorithm is performed in the function `stopToDigitalSim.m`. A template file can be found in:

```
./ccm/Mspice/stopToDigitalSim_Template.m
```

Copy this file to your synchronizer directory and rename the file to `stopToDigitalSim.m`. The function is called within the `nestedBisection testbench` as follows:

```
[value,isterminal,direction] = stopToDigitalSim(t,Vin,vDot,digitalSimOptions)
```

This function implement the `event` function for the ode solver, documentation can be found at `https://www.mathworks.com/help/matlab/ref/odeset.html#d120e857760`.

1. *t* the time point t of the numerical integration
2. *Vin* the voltage state of the synchronizer at time t.
3. *vDot* is the time derivative of the synchronizer at time t.
4. *digitalSimOptions* is a struct which the user can include any relevant information pertaining to how to determine when to conclude the current round of the nested bisection algorithm.

If no stopping criteria can be determined, return *value = -1*, *isterminal = 0* and *direction = 1*.

6.2.5 nestedBisectionStop.m

The user provided `nestedBisectionStop.m` function is used to give the user the flexibility to choose what criteria to use to terminate the nested bisection algorithm when the `simulate` method is called.

The function has the following signature:

```
stop = nestedBisectionStop(tbOptions,numStatesPerCCT,V,tCrict,t)
```

1. `stop`: A boolean variable which is set to `true` if the user defined stopping criteria is met. When se to `true` the nested bisection algorithm will stop.
2. *tbOptions*: The testbench options are made available as a means to transfer relevant information to determine when the terminating criteria to the algorithm is met.

3. *numStatesPerCCT*: The number of states per simultaneous simulation is provided to subdivisde *V* into individual circuit states.

4. *V*: The total voltage state of all simultaneously simulated circuits.

5. *tCrit*: The user specified critical time by which the synchronizer must have a stable well defined digital value on its output.

6. *t*: The time points of the current numerical integration run.

6.2.6 dV_ode

Same as testbench dV_ode except for the application of d_{in} to the input voltage source of the synchronizer.

```
myNBTestbench.dV_ode(t,v_ode)
```

$d_{in} = v(t, t_{in})$ of a voltage source that is parametrized by t_{in} (such as vtanhDelay see Section 5.4). The transition times t_{in} are equally divided by the number of parallel circuits being simulated numParallelCCTs between the current input window time interval $[t_{hi}, t_{lo}]$.

6.2.7 findClockEdges

This function computes the number of clock edges specified in tbOptions and returns a list of times at which the 50% level of the input clock source occurs. The input *t* are the time points to sample the input clock source.

```
clockEdgeTimes = myNBTestbench.findClockEdges(t)
```

6.2.8 findNextBracket (protected)

This function returns two indices. One which correspond to the synchronizer circuit whose trajectory resulted in a settle high output and the other as the one whose output settled low. These indices have the tbOption.stepOut' applied for robustness.

```
index = myNBTestbench.findNextBracket(Vin,t,edgeTimes)
```

1. *Vin*: The input voltage state of the numerically integrated trajectories of all simultaneously simulated synchronizer circuits

2. *t*: The associated time points for the numerically integrated results]item *edgeTimes*: The list of clock edges corresponding to the results obtained from findClockEdges (see above)

6.2.9 findNextStartStatesAndTime (protected)

This function returns:

1. starTime: the new start time for the next bisection epoch

2. `newStates`: the new set of initial conditions

3. `VH` and `VL`: the circuit state at the end of the current epoch corresponding to the synchronizer circuit trajectories which had its output settle high and low respectively

4. `diffStates`: the difference between the fit polynomial model and the numerically integrated trajectories

5. `timeInt`: the time interval where the linearity assumption of characterizing all synchronizer circuits is valid.

```
[starTime,newStates,VL,VH,diffSTates,timeInt] = ···
myNBTestbench.findNextStartStatesAndTime(Vin,t,index)
```

1. *Vin*: is the numerically integrated voltage state

2. *t*: is the time points associated with the numerically integrated voltage state

3. *index*: is the indices that select which synchronizer circuit settled high and which one settled low – determined by `findNextBracket`.

6.2.10 `bisectionPlotSetup` (protected)

This function creates and initializes the plots used during the nested bisection algorithm to provide the user with visual queues on the progress of the algorithm. (Future work is to provide plotting options via `tbOptions`). The return is a set of plot handles used as the algorithm progress: `plotHandles`.

```
plotHandles = myNBTestbench.bisectionPlotSetup(tspan,tCrit,tOff)
```

1. *tspan*: is the time span specified by the user that the nested bisection algorithm is to simulate to: $t \in [t_{start}, t_{end}]$

2. *tCrit*: the user specified time by which the synchronizer needs to be settled to stable digital values

3. *tOff*: the offset time which shifts time such that $t = 0$ at the 50% voltage level of the first edge of the input clock

6.2.11 `bisectionPlotV` (protected)

This function plots all simulated voltage signals for the `numParallelCCTs` being simulated on the appropriate plot provided by *plotHandles* over the numerically integrated time points *t* and state variable *Vin*. The nodes shown by default are those specified by the input `mask` in the constructor.

```
myNBTestbench.bisectionPlotV(plotHandles,t,Vin)
```

6.2.12 `bisectionPlotBeta` (protected)

This function plots the numerical estimation of *beta* over the linear time interval *tbeta* on the appropriate plot provided by *plotHandles*.

```
myNBTestbench.bisectionPlotBeta(plotHandles,tbeta,beta)
```

6.2.13 `vfull` (protected)

This function overrides `vfull` of the `testbench` class. It creates the full voltage vector which includes the evaluation of `voltage sources` as described in `testbench`, however, in this version the input source for $d_{in} = v(t, t_{in})$, thus t_{in} for each ith synchronizer `circuit` gets the ith value for t_{in} which is equally split on the current interval $t_{in} \in [t_{hi}, t_{lo}]$.

```
vfull = myNBTestbench.vfull(t,v_ode)
```

6.2.14 `setInterval` (protected)

This function sets the attribute of the `nestedBisection testbench` to the specified new interval *newInterval* in the form $[t_{hi}, t_{lo}]$.

```
myNBTestbench.setInterval(newInterval)
```

6.3 `nestedBisectionWithKick.m` class

This subclass of a `nestedBisection testbench` only overrides the `simulate` function. This `testbench` provides the ability for the user to specify at time t_{kick}, a set of selected nodes $V_{kick} \subseteq V$ which get multiplied by a perturbation ε, i.e. $V(kick) = \varepsilon V(kick)$.

```
bisectionKickData = myNB_KickTestbench.simulate(filename,tspan,tCrit,···
                    kickData,v0)
```

1. *filename*: a string which saves the nested bisection `bisectionRuns` simulation data to `'filename.mat'`
2. *timeSpan*: the overall time span the numerical integration is to take place.
3. *tCrit*: the time by which the synchronizer needs to just meet the user defined `threshold` for digital levels. Also used as a check to determine if the `nestedBisectionStop.m` (see Section 6.2.5) function is to be called to terminate the algorithm.
4. *kickData*: a `struct` which must contain the following:
   (a) *kickTime*: the time t_{kick} at which the 'kick'/perturbation occurs
   (b) *kickPercentage*: the amount the selected nodes will be 'kicked'/perturbed at time t_{kick}
   (c) *kickNodes*: the nodes that are affected by the 'kick'/perturbation
5. *v0*: initial voltage state to be applied to the synchronizers.
   (optional - if not supplied *v0* is a vector of all zeros)

6.4 `nestedBisectionAnalysis.m` class

The `nestedBisectionAnalysis testbench` subclass uses the results of the `nestedBisection simulate` results to construct a small-signal linear model which is used to construct a time-to-voltage gain performance metric as described in [15–17]. Found in this section are the methods which are used to provide the pieces discussed in those results.

The AD package found in [18] is required to make use of the methods found in this class. The separation of this class from the `nestedBisction` class means that a user can still run the nested bisection algorithm without using the methods found in this class.

Section 6.4.15 and Section 6.4.14 describe necessary function definitions used in the method `dGdw` and explained in greater detail in Section 4.

6.4.1 `nestedBisectionAnalysis`

A `nestedBisectionAnalysis testbench` has the following attributes, they are very similar to a `nestedBisection` class:

1. *`inputSource`*: the input source parametrized with by t_{in} which changes when the d_{in} transition occurs
2. *`clockSource`*: the clock source for the synchronizers
3. *`inputSourceIndex`*: the index which corresponds to the input source among the list of sources for the synchronizers
4. *`outputNode`*: the final output node that needs to be digitally stable by t_{crit}
5. *`defaultModelAD`*: the default AD `model` for the analysis
6. *`synchronizerCCTMap`*: synchronizer `circuit` device mapping information/vectorized attributes for the `circuit` definition (protected)
7. *`syncMatrixMap`*: a sparse matrix mapping for all `numParallelCCTs` being simulated (protected)

To create a `nestedBisectionAnalysis` class:

```
myNBAnalyis = nestedBisctionAnalysis(circuit,ports,sources,inputSource,···
              clockSource,metaResOutput,model,process,options)
```

1. *`circuit`*: is a synchronizer circuit which has an input d_{in} which is parametrized by the value t_{in}.
2. *`ports`*: are the synchronizer's *ports* excluding d_{in} input as a cell array in the order that they are listed, e.g. {`mySync.vdd,mySync.gnd,mySync.clk,mySync.reset`}.
3. *`sources`*: are the sources input as a cell array connecting to the synchronizer's *ports* in the order in which they appear, e.g. {`Vdd,Gnd,Clk,Reset`}.
4. *`inputSource`*: is the `circuit`'s input source index and `voltage source` for the synchronizer's d_{in} signal, passed as a cell array. e.g. {`sync.d,din`}

5. *clockSource*: is the clock source for the synchronizer, used to determine when the clock edges occur to switch which node in *mask* to consider.
6. *metaResOutput*: is the node in the synchronizer that must be digitally stable by t_{crit} in order to have a successful synchronization event.
7. *model*: is the default `model` that the `testbench` uses for `simulation`. (optional - if not supplied the default `model` is the *MVS* `model`)
8. *process*: is the `process` for the `testbench`'s *model*. (optional - if not supplied the default is the *PTM 45nmHP* `process`)
9. *options*: is the `option` object that determine parameters for the `testbench` at simulation time. (optional see default `tbOptions` when not supplied below).

The default `tbOptions` in constructing a `nestedBisectionAnalysis testbench` are the same as those found in a `nestedBisection testbench` with the following changes/additions:

1. *isAD* - `true`
2. *'integratorOptions'*: `odeset('RelTol',1e-6,'AbsTol',1e-8,'InitialStep',0.5e-12)`
3. *'betaSimOptions'*: a `struct` with the following:
   (a) *'minSimTime'* - 6e-10
   (b) *'threshold'* - 1e6
4. *'integratorOptionsBeta'*: `odeset('RelTol',1e-6,'AbsTol',1e-6,'InitialStep',0.5e-12)`

6.4.2 nested bisection analysis options: `tbOptions`

Listed here are the unique options which are not listed in Section 6.1.1 or Section 6.2, those listed in those `testbenches` also apply here. The `struct` follows the `<'string',value>` pair as described in the `testbench options`.

New in the `tbOptions` is the *'betaSimOptions'* and *'integratorOptionsBeta'* which are analogous to the `digitalSimOptions` in the `nestedBisection` class. They are used if the `betaRenorm Event` function is enabled. The variable *'threshold'* determines how large $\beta(t)$ can get before it is re-normalized. This is to prevent numerical issues with $\beta(t)$ an exponentially growing function in time.

6.4.3 `dbeta_ode`

This function computes the time derivative $\dot{\beta(t)}$ `dbeta` used in numerically integrating this function to compute $\beta(t)$:

`dbeta = myNBAnalyis.dbeta_ode(`*Jac*`,`$\frac{\partial f}{\partial t_{in}}$`,`$\beta$`)`

1. *Jac*: the Jacobian of the synchronizer at time t
2. $\frac{\partial f}{\partial t_{in}}$: the derivative of $f = C^{-1}I$ w.r.t. t_{in} at time t
3. β: the numerically integrated result for β at time t

6.4.4 `dwdt_ode`

This function computes the time derivative $\dot{w}$ of an intermediate quantity $w(t)$ which is related to $u(t)$, where ***Jac*** and ***w_ode*** are the Jacobian and numerically integrated value of w at time t.

`dwdt = myNBAnalyis.dwdt_ode(`***Jac***`,`***w_ode***`)`

6.4.5 `gainSync`

This function computes the synchronizer gain `g`, its time derivative `dgdt` and the homogeneous part of the differential equation described in [15–17] `lambda`.

`[g,dgdt,lamba] = myNBAnalyis.gainSync(`***u(t)***`,`***Jac(t)***`,`$\beta(t)$`,`$\frac{\partial f(t)}{\partial t_{in}}$`)`

1. ***u(t)***: is the normalized unit vector that projects β onto the nodes of interest at time t
2. ***Jac(t)***: the Jacobian of the circuit at time t
3. $\beta(t)$: the synchronizer's time to voltage state gain β at time t
4. $\frac{\partial f(t)}{\partial t_{in}}$: the derivative of $f = C^{-1}I$ w.r.t. t_{in} at time t

6.4.6 `splineBisection`

This function takes the results of the nested bisection algorithm ***bisectionData*** and user defined ***tCrit*** to compute a splined object $V_{meta}(t)$ so that it can be evaluated at any time points for $t \in [0, t_{eola}]$. The combined time points t and the end of linear analysis time t_{eola} are also returned to set bounds on the numerical integration.

WARNING: Extrapolation of $V_{meta}(t)$ yields non-nonsensical data.

`[`$V_{meta}(t)$`,`t`,`t_{eola}`] = myNBAnalyis.splineBisection(`***bisectionData***`,`***tCrit***`)`

6.4.7 `getVfull`

This function provides a means for the user to get the full voltage vector `vfull` from the *private* method `vfull` of the synchronizer `circuit` for a specified input $V_{meta}(t)$ and associated ***timePoints*** and a specified t_{in}.

`vfull = myNBAnalyis.getVfull(`***timePoints***`,`t_{in}`,`$V_{meta}(t)$`)`

6.4.8 `computeJacVdot`

This function returns:

1. `Jac`: The Jacobian of the `circuit` state
2. $\frac{\partial f}{\partial t_{in}}$: the derivative of f w.r.t. t_{in}
3. $\dot{V}$: the time derivative of the synchronizer `circuit`

4. $\frac{\partial I}{\partial V}|_{Dev}$: the derivative of all `circuit` element currents with respect to the synchronizer voltage state V

5. C_{Dev}: the capacitance matrix associated with each `circuit` element of the synchronizer

for a specified time `t`, t_{in} and metastable voltage state V_{meta}. These quantities are used in additional analysis while also providing means to compute individual component Jacobian contributions.

This function makes calls to `computeVdot` and `computJac` to compute the outputs.

`[Jac,`$\frac{\partial f}{\partial t_{in}}$`,`$\dot{V}$`,`$\frac{\partial I}{\partial V}|_{Dev}$`,`$C_{Dev}$`] = myNBAnalyis.computeJacVdot(t,`$t_{in}$`,`$V_{meta}$`)`

6.4.9 `computeVdot`

This function computes:

1. $\dot{V}$: The time derivative of the synchronizer `circuit`

2. `C`: The `circuit`'s overall capacitance matrix

3. $I_{mathitDev}$: The current flowing in all devices

4. `capParams`: The associated capacitor parameters for all devices.

for a specified time `t`, t_{in} and metastable voltage state V_{meta}.

`[`$\dot{V}$`,C,`I_{Dev}`,capParams] = myNBAnalyis.computeVdot(t,`t_{in}`,`V_{meta}`)`

6.4.10 `computeJac`

This function computes:

1. `Jac`: the Jacobian of the synchronizer `circuit`

2. $\frac{\partial f}{\partial t_{in}}$: the derivative of f w.r.t. t_{in}

3. $\frac{\partial I}{\partial V}|_{Dev}$: the derivative of all `circuit` element currents with respect to the synchronizer voltage state V

4. C_{Dev}: the capacitance associated with every device in the `circuit` definition

and makes use of the results form `computeVdot` to avoid re-calculating results where possible.

`[Jac,`$\frac{\partial f}{\partial t_{in}}$`,`$\frac{\partial I}{\partial V}|_{Dev}$`,`$C_{Dev}$`] = myNBAnalyis.computeJac(t,`$t_{in}$`,`$V_{meta}$`,C,I,capParams)`

1. `t`: the time point t

2. t_{in}: the time d_{in} changes

3. V_{meta}: the metastable voltage state of the synchronizer `circuit` at time t

4. *C*: the pre-computed capacitance matrix from `computeVdot`
5. *I*: the pre-computed device currents from `computeVdot`
6. *capParams*: the pre-computed capacitance parameters from `computeVdot`

6.4.11 computeMapMatrixDevice

This function returns the connectivity matrix N_{Dev} for all devices that are connected to nodes `in` and `out`.

This provides the user the ability to multiply this connectivity matrix N_{Dev} by the total device Jacobian to select the Jacobian elements that correspond to the devices in question.

N_{Dev} = `myNBAnalyis.computeMapMatrixDevice(`*in*`,`*out*`)`

6.4.12 computeMapMatrixDeviceTx

This function returns the connectivity matrix $N_{Tx_{Dev}}$ for all transistors of *type* (pmos or nmos) connected to nodes *in* and *out* which allows the user to select individual transistor types connected to nodes of interest.

This allows the user to determine the Jacobian elements that correspond to the device(s) (nmos or pmos) in question.

$N_{Tx_{Dev}}$ = `myNBAnalyis.computeMapMatrixDeviceTx(`*in*`,`*out*`,`*type*`)`

6.4.13 dGdw

This function requires user provided functions `populateADOptimizationVar.m` and `expandW.m`. The function returns the gradient of the synchronizer gain $g(t)$ with respect to the user selected variables w at time t_{crit} - `dGdw_tCrit`. In addition this function returns the intermediate results `gamma` and associated time points `t`.

The function `populateADOptimizationVar.m` and `expandW.m` are called internally to the call to this method. They are inverses of each other and details are found in Section 6.4.14 and Section 6.4.15 respectively.

`[dGdw_tCrit,gamma,t] = myNBAnalyis.dGdw(`$V_{meta}(t)$`,`*Jac(t)*`,`$\frac{\partial f(t)}{\partial t_{in}}$`,`$t_{in}$`,`$\cdots$
$\beta(t)$`,`*timeSpan*`,`*uVcrit*`)`

1. $V_{meta}(t)$: the splined synchronizer `circuit` metastable voltage state
2. *Jac(t)*: the splined Jacobian of the synchronizer `circuit`
3. $\frac{\partial f(t)}{\partial t_{in}}$: the splined derivative of f w.r.t. t_{in}
4. t_{in}: the time at which d_{in} of the synchronizer effectively transitions
5. $\beta(t)$: the splined synchronizer β vector
6. *timeSpan*: the time span for the simulation i.e. $[0, t_{eola}]$
7. *uVcrit*: the user specified normalized unit vector which determines the performance metric towards metastability resolution at time t_{crit}

6.4.14 `populateADoptimizationVar.m`

This function returns the vector of AD variables (`hessian` variables), `adVar` and the number of variables `numVar`. The number of AD variables `numVar` must be $|w_{opt}| \leq |w|$ and used to compute $\frac{\partial G_{sync}}{\partial w}$ where the set of transistor widths to consider in the optimization w_{opt} is defined in this function by the user.

This user defined function allows the user to impose constraints as described in Section 4.

`[adVar,numVar] = populateADoptimizationVar(`*`voltageState`*`,`*`nDevices`*`,`*`pDevices`*`)`

1. *`voltageState`*: is the voltage state of the synchronizer
2. *`nDevices`*: is the set of `nmos` device widths in the synchronizer `circuit` definition
3. *`pDevices`*: is the set of `pmos` device widths in the synchronizer `circuit` definition

6.4.15 `expandW.m`

This function expands the subset *`wOpt`* back into the full vector `w`. The function can be thought of as undoing the contraction that `populateADoptimizationVar.m` performs. It is the user's responsibility to determine how to expand w_{opt} back to w.

`w = expandW(`*`wOpt`*`)`

6.4.16 `gamma_ode`

This function computes the time derivative `gammaDot` which computes $\frac{\partial \dot{\beta}}{\partial w}$. This function is used to numerically integrate `gammaDot` to obtain `gamma`.

`gammaDot = myNBAnalyis.gamma_ode(`*`t`*`,`V_{meta}`,Jac,`$\frac{\partial f}{\partial t_{in}}$`,`$\beta$`,`*`gamma`*`,`$t_{in}$`)`

1. *`t`*: the current time point t during integration
2. V_{meta}: the synchronizer metastable voltage state at time t
3. *`Jac`*: the synchronizer Jacobian at time t
4. $\frac{\partial f}{\partial t_{in}}$: the synchronizer derivative of f w.r.t. t_{in}
5. β: the vector β at time t
6. *`gamma`*: the numerically integrated value of gamma at time t
7. t_{in}: the time at which d_{in} changes

6.4.17 `vfull` (private)

Implements the same function as discussed in the `testbench` but turns the resulting vector into an AD variable.

6.4.18 `betaRenorm` (private)

This function is used if in the `tbOptions.integratorOptionsBeta` includes in its `odeset` the field `'Event',@this.betaRenorm`. (see details on Event function in MATLAB at `https://www.mathworks.com/help/matlab/ref/odeset.html#d120e857760`)

`[value,isTerminal,direction] = myNBAnalyis.betaRenorm(`t`,`β`)`

Within the `betaSimOptions` is a threshold which if exceed causes this function to terminate the integration so that the user can re-normalize $\beta(t)$ to prevent numerical issues as $\beta(t)$ is a function which grows exponentially.

7 MSPICE Models

v2.0 of MSPICE provides three transistor `model` devices:

1. `MVS model` adapted from the work found in [9, 11, 14, 20] to be implemented such that an AD friendly version could be created: `MVS AD model`
2. `EKV model` adapted from the work found in [2–5] to produce a simplified version for MSPICE
3. `interpModel` from Yan's thesis work in [21]

A junction capacitance model for transistors is also provided based on the [6] model outlined in Section 7.3.

Further details on the models and the results are discussed in [17] and Section 2.1.2.

To facilitate the addition of new `model` classes to MSPICE a `modelDictionary` class and a `model` interface have been created. This provides a skeleton template to use when adding new model cards. New methods can be added to the `model` interface while still preserving old functionality.

Their access in a `testbench` class is made straight forward and also encourages making new `testbench` subclasses for particular applications.

This section describes the `modelDictionary` class, the `model` interface and the sketch of how the models available are used.

7.1 Model Dictionary class

The `modelDictionary` class is the bridge between `models` and the `testbench` class. A `testbench` is instantiated with a `modelDictionary` which is the gate keeper to user specified models such as *interp*, *MVS* and *EKV* `models`. A `testbench` retrieves the specific `model` by using the `getModel` method which configures the `model` with the user specified parameters.

To add a new model to the `modelDictionary` simply add a case in the `getModel` method with its appropriate configuration. The `modelDictionary.m` file is found in:

`./ccm/models/modelDictionary.m`

To simplify the passing of model specific parameters a `struct` with `<'key',value>` pairs is the convention used as the universal means to pass information to configure the specific `models`.

The constructor to the `modelDictionary` takes no arguments:

```
myDict = modelDictionary
```

Currently in v2.0 of MSPICE the `modelDictionary` has definitions for:

1. *'interp'*
2. *'MVS'*
3. *'MVS AD'*
4. *'EKV'*
5. *'EKV AD'*

The AD version of the models are separated so that `testbenchs` that require AD functionality such as `nestedBisectionAnalysis` are able to do so without hindering the use of the non AD versions. The *EKV* `model` has no implementation difference as it is inherently constructed such that it is AD friendly.

7.1.1 getModel

The `getModel` function returns the user selected configured `model`:

```
myModel = myDict.getModel(modelName,modelProcess,modelOptions)
```

1. *modelName*: is the name of the model to be used such as *interp*, *MVS* or *EKV*
2. *modelProcess*: is the process used for that particular model i.e. *PTM 45nmHP*
3. *modelOptions*: is a `struct` with the options that are to be applied for the model, such as *temp* to set the temperature

The configuration of the model is performed internally by calling the method `getModelConfig`.

7.1.2 getModelConfig

The `getModelConfig` function configures the `model` in question with the associated `modelOptions` and `process` provided. Each `model` is responsible for implementing its own configuration routine which is aware of the *processes* that are available.

The naming convention for a `model` configuration functions is:

```
config = ccm_cfg_<insert model name here>
```

where <insert model name here> is the name of the model in question. The configuration functions will return a struct with the information for the model and its associated process. An error will be thrown if the information is not available.

The getModelConfig has the following signature:

```
modelConfigHandle = myDict.getModelConfig(model,process,options)
```

1. *model*: the name of the model such as *interp*, *MVS* or *EKV*
2. *process*: is the model process such as *PTM 180nm* or *PTM 45nmHP*
3. *options*: are options in struct form to modify the configuration defaults, such as *temp*

7.2 Model interface

The model class is outlined in the interface model.m found in:

```
./ccm/models/model.m
```

The constructor of a model:

```
myModel = model(modelName,modelParams)
```

Where *modelName* is the name of the model and *modelParams* is the the struct that contains the model parameters. The user is free to choose the <'key',value> pairs since the user is free to implement the model class interface as they see fit.

Found in this section are the methods that are used in the testbench class at simulation time. Additional specialized functions can be added for new and specific testbench subclasses. The list included here are the model methods used in the testbenchs described in Section 6 which are overridden specifically for the nmos and pmos devices.

7.2.1 I

This function returns the current iDev going into each terminal of the device in question and the capacitance parameters for the device cParams as it is often the case that the capacitance is related to the computation, such as computing charge in transistor devices relates to the capacitance. Its call signature is:

```
[iDev,cParams] = myModel.I(deviceType,deviceParams,Vin,tbOptions)
```

1. *deviceType*: the type of device, currently supported (or scaffolding in place) in v2.0 of MSPICE:
   (a) *pmos*: calls iPMOS
   (b) *nmos*: calls iNMOS

(c) *vsrc*: currently returns no current

(d) *inductor*: calls `iL`

(e) *capacitor*: calls `iC`

(f) *resistor*: calls `iR`

2. ***deviceParams***: the device parameters

3. ***Vin***: the voltage vector for the terminal nodes of the device

4. ***tbOptions***: additional options

7.2.2 `IdevRes`

This function returns the residual `ires` of the current of the device and the Jacobian `J` of the device in question w.r.t. the variables in ***Vin***. Note: the residual is only useful when an interative solver is involved to compute the current of the device.

```
[ires,J] = myModel.IdevRes(deviceType,deviceParams,Vin,Idev,numCircuitNodes,options)
```

1. ***deviceType***: the type of device, currently supported (or scaffolding in place) in v2.0 of MSPICE:

   (a) *pmos*: calls `iresPMOS`

   (b) *nmos*: calls `iresNMOS`

   (c) *vsrc*: currently returns nothing

   (d) *inductor*: calls `iresL`

   (e) *capacitor*: calls `iresC`

   (f) *resistor*: calls `iresR`

2. ***deviceParams***: the device parameters

3. ***Vin***: the voltage vector for the terminal nodes of the device

4. ***Idev***: the result from calling `Idev = myModel.I(type,params,Vin,opts)`

5. ***numCircuitNodes***: the number of nodes in the `circuit` in question to compute the Jacobian properly

6. ***options***: additional options

7.2.3 `C`

This function returns the capacitance `c` for the associated device. The capacitance parameters computed in `I` are best added to the input device parameters ***deviceParams***.

```
c = myModel.C(deviceType,deviceParams,Vin,options)
```

1. *deviceType*: the type of device, currently supported (or scaffolding in place) in v2.0 of MSPICE:
   (a) *pmos*: calls `cPMOS`
   (b) *nmos*: calls `cNMOS`
   (c) *vsrc*: currently returns nothing
   (d) *inductor*: calls `cL`
   (e) *capacitor*: calls `cC`
   (f) *resistor*: calls `cR`
2. *deviceParams*: the device parameters
3. *Vin*: the voltage vector for the terminal nodes of the device
4. *options*: additional options

7.2.4 iPMOS

Function to be overridden by the particular `model`. Returns the current going into (or out of) the terminals of a `pmos` device(s) and any associated capacitance parameters.

`[i,cParams] = myModel.iPMOS(`*deviceParams*`,`*Vin*`,`*options*`)`

7.2.5 iNMOS

Function to be overridden by the particular `model`. Returns the current going into (or out of) the terminals of a `nmos` device(s) and any associated capacitance parameters.

`[i,cParams] = myModel.iNMOS(`*deviceParams*`,`*Vin*`,`*options*`)`

7.2.6 iL

Place holder function for now. Implement default functionality to return current into nodes of `inductor(s)` and any associated capacitance parameters.

`[i,cParams] = myModel.iL(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.7 iC

Place holder function for now. Implement default functionality to return current into nodes of `capacitor(s)` and any associated capacitance parameters.

`[i,cParams] = myModel.iC(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.8 iR

Place holder function for now. Implement default functionality to return current into nodes of `resistor(s)` and any associated capacitance parameters.

`[i,cParams] = myModel.iR(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.9 cPMOS

Function to be overridden by the particular `model`. Returns the capacitance for all nodes (or inter node capacitances if the `capModel` being used is `'full'`) for the `pmos` device(s).

`c = myModel.cPMOS(`*deviceParams*`,`*Vin*`,`*options*`)`

7.2.10 cNMOS

Function to be overridden by the particular `model`. Returns the capacitance for all nodes (or inter node capacitances if the `capModel` being used is `'full'`) for the `nmos` device(s).

`c = myModel.cNMOS(`*deviceParams*`,`*Vin*`,`*options*`)`

7.2.11 cL

Place holder function for now. Implement default functionality to return capacitance on nodes of `inductor(s)` and inter node capacitance if necessary.

`c = myModel.cL(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.12 cC

Place holder function for now. Implement default functionality to return capacitance on nodes of `capacitor(s)` and inter node capacitance if necessary.

`c = myModel.cC(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.13 cR

Place holder function for now. Implement default functionality to return capacitance on nodes of `resistor(s)` and inter node capacitance if necessary.

`c = myModel.cR(`*deviceParams*`,`*Vin*`,`*options*`)`

Or override with particular model.

7.2.14 iresPMOS

Function to be overridden by the particular `model`. Returns the residual current (if applicable) and the Jacobian for `pmos` device(s) with respect to the variables in `Vin`.

```
[ires,J] = myModel.iresPMOS(deviceParams,Vin,I,numCircuitNodes,options)
```

7.2.15 iresNMOS

Function to be overridden by the particular `model`. Returns the residual current (if applicable) and the Jacobian for `nmos` device(s) with respect to the variables in `Vin`.

```
[ires,J] = myModel.iresNMOS(deviceParams,Vin,I,numCircuitNodes,options)
```

7.2.16 iresL

Place holder function for now. Implement default functionality to return the residual (if applicable) and the Jacobian of `inductor(s)` with respect to the variables in `Vin`.

```
[ires,J] = myModel.iresL(deviceParams,Vin,I,numCircuitNodes,options)
```

Or override with particular model.

7.2.17 iresC

Place holder function for now. Implement default functionality to return the residual (if applicable) and the Jacobian of `capacitor(s)` with respect to the variables in `Vin`.

```
[ires,J] = myModel.iresC(deviceParams,Vin,I,numCircuitNodes,options)
```

Or override with particular model.

7.2.18 iresR

Place holder function for now. Implement default functionality to return the residual (if applicable) and the Jacobian of `resistor(s)` with respect to the variables in `Vin`.

```
[ires,J] = myModel.iresR(deviceParams,Vin,I,numCircuitNodes,options)
```

Or override with particular model.

7.3 Model cards

The `models` available in MSPICE are outlined in this section. These `model` classes override the necessary functions outlined in the `model` interface to provide current, Jacobians and capacitances for the transistor models that they describe.

These `models` have their own user defined functions to compute the current (typically I_{ds}) for MOS devices.

The `testbench` makes calls to the `model` methods with the assumption that they are vectorized. If the model does not support vectorization it is the user's responsibility to "de-vectorize" the data in their implementation of the `model` methods.

The details on how to fit the `models` to collected transistor IV characteristic curves is not described in this section.

7.3.1 `junctionDrainCap.m`

The `junctionDrainCap.m` function provides a higher fidelity capacitance model by adding the necessary junction capacitance found in transistors, i.e. modelling C_{db} (drain-body or source-body) capacitance. In v2.0 of MSPICE, this model is directly implemented from the BSIM model [6] manual.

```
Cj = junctionDrainCap(Vdb,params)
```

where ***Vdb*** is the drain-body voltage and ***params*** are the necessary parameters from the original transistor model card for the BSIM model of SPICE.

This model is only used when considering a `capModel = 'full'`.

This function is found in:

```
./ccm/models/junctionDrainCap.m
```

7.3.2 `interpModel.m`

The `interpModel` is the one used in version 1.0 of the coho package and fits data collected with the `'PTM 180nm'` process. At this time no AD version of the model is supported. The `interModel` and configuration function are found in:

```
./ccm/models/interpModel.m
./ccm/models/ccm_cfg_interp.m
```

The data and particular functions to evaluate MOS devices are found in:

```
./ccm/models/Interp
```

7.3.3 `ekvModel.m`

The `ekvModel` is a simplified version of the work found in [2–5] and is fit with data collected with the `'PTM 45nmHP'` process. The simplified `ekvModel` found in MSPICE is inherently AD friendly thus no modifications are necessary. The `ekvModel` and configuration function are found in:

```
./ccm/models/ekvModel.m
./ccm/models/ccm_cfg_ekv.m
```

The data and particular functions to evaluate MOS devices are found in:

```
./ccm/models/EKV
```

7.3.4 mvsModel.m

The mvsModel is an adapted version of the work found in [9, 11, 14, 20] and is fit with data collected with the 'PTM 45nmHP' process. Only the capModel = 'gnd' is available with this model. This version does not provide a means to compute the Jacobian. The mvsModel and configuration function are found in:

```
./ccm/models/mvsModel.m
./ccm/models/ccm_cfg_mvs.m
```

The data and particular functions to evaluate MOS devices are found in:

```
./ccm/models/MVS
```

7.3.5 mvsADModel.m

The mvsModel is an adapted version of the work found in [9, 11, 14, 20] and is fit with data collected with the 'PTM 45nmHP' process. The AD version provides the additional functionality to work with the capModel = 'full' and provides functionality to compute the Jacobian.The ekvModelAD and configuration function are found in:

```
./ccm/models/ekvModelAD.m
./ccm/models/ccm_cfg_ekvAD.m
```

The data and particular functions to evaluate MOS devices are found in:

```
./ccm/models/MVS
```

8 MSPICE Library contents

List of all library contents

8.1 Circuits

IRO.m
JAMB_TX_TEST.m
PASSGATE_LATCH_TEST.m
RRO.m
TEMPLATE.m
THREE_FLOP_PG_SYNC.m
THREE_RING_OSC.m
TWO_FLOP_BUFFERED_JAMB_SYNC.m
TWO_FLOP_JAMB_SYNC.m
TWO_FLOP_PG_SYNC.m
TWO_FLOP_PG_SYNC_SCAN.m
TWO_FLOP_ROBUST_JAMB_SYNC.m
TWO_FLOP_STRONG_ARM_SYNC.m
VSRC_TEST.m
A work in progress:
TWO_FLOP_PSEUDO_NMOS_SYNC.m
TWO_FLOP_PSEUDO_NMOS_SYNC_TEST.m

8.2 Components

CELEMENT.m
FF.m
INV.m
INV4.m
JAMB_FF.m
JAMB_FF_BUFFERED.m
JAMB_LATCH.m
LATCH.m
NAND.m
NAND2.m
NOR.m
PASSGATE.m
PASSGATE_FF.m
PASSGATE_FF_SCAN.m
PASSGATE_LATCH.m
PASSGATE_LATCH_SCAN.m
ROBUST_JAMB_FF.m
ROBUST_JAMB_LATCH.m
STRONG_ARM_FF.m
STRONG_ARM_LATCH.m
A work in progress:
PSEUDO_NMOS_FF.m
PSEUDO_NMOS_LATCH.m

8.3 Circuit leaves

resistor.m
capacitor.m
nmos.m
pmos.m
Not implemented yet:
inductor.m

8.4 Voltage Sources

vpulse.m
vsin.m
vsrc.m
vsrcLinear.m
vtanh.m
vtanhClock.m
vtanhDelay.m

9 Future Work and Improvements

Here are a list of improvements and future work.

1. Currently `resistor` and `capacitor` are `circuit` classes but should be implemented as `coho_leaf` classes
2. Add the definition for a `inductor coho_leaf`
3. Add default current, voltage and capacitance returns in the `model` class interface for `resistors`, `capacitors` and `inductors`
4. The `EKV model` currently has poor performance at low V_{ds} biasing, add term to improve this
5. A lot of code from version 1 of coho present in the `circuit` class which should be moved into the `interp model` using the `model` interface class
6. Create a diode `coho_leaf`
7. Create a simpler `junctionCapacitance` model to be used not based on the full blown BSIM model
8. Create a section or separate document/tutorial on fitting your own *EKV*, *MVS* or *interp* model from transistor data

References

[1] L. M. engelhardt. *version XVII*. Linear Techology Corporation - now part of Analog Devices, 2019.

[2] C. Enz. An MOS transistor model for RF IC design valid in all regions of operation. *IEEE Transactions on Microwave Theory and Techniques*, 50(1):342–359, Jan 2002.

[3] C. C. Enz. A short story of the EKV MOS transistor model. *IEEE Solid-State Circuits Society Newsletter*, 13(3):24–30, Summer 2008.

[4] C. C. Enz, F. Krummenacher, and E. A. Vittoz. An analytical MOS transistor model valid in all regions of operation and dedicated to low-voltage and low-current applications. *special issue of the Analog Integrated Circuits and Signal Processing Journal on Low-Voltage and Low-Power Design*, 8:83–114, 1995.

[5] C. C. Enz and E. A. Vittoz. Charge-based MOS transistor modeling - the EKV model for low-power and RF IC design. 2006.

[6] B. Group. Berkeley short-channel IGFET model (BSIM) group, Feb 2017.

[7] HSPICE. *the gold standard for accurate circuit simulations*. Synopsis Inc., Mountain View, California, 2019.

[8] N. Integration and M. N. Group. Predictive technology model, June 2011.

[9] A. Khakifirooz, O. M. Nayfeh, and D. Antoniadis. A simple semiempirical short-channel MOSFET current-voltage model continuous across all regions of operation and employing only physical parameters. *IEEE Transactions on Electron Devices*, 56(8):1674–1680, Aug 2009.

[10] T. Kobayashi, K. Nogami, T. Shirotori, Y. Fujimoto, and O. Watanabe. A current-mode latch sense amplifier and a static power saving input buffer for low-power architecture. In *1992 Symposium on VLSI Circuits Digest of Technical Papers*, pages 28–29, June 1992.

[11] M. S. Lundstrom and D. A. Antoniadis. Compact models and the physics of nanoscale FETs. *IEEE Transactions on Electron Devices*, 61(2):225–233, Feb 2014.

[12] MATLAB. *version 9.3.0.713579 (R2017b)*. The MathWorks Inc., Natick, Massachusetts, 2017.

[13] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stehpany, and S. C. Thierauf. A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, Nov 1996.

[14] S. Rakheja and D. Antoniadis. MVS nanotransistor model (silicon), Dec 2015.

[15] J. Reiher and M. R. Greenstreet. Optimization and comparison of synchronizers. In *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2020.

[16] J. Reiher, M. R. Greenstreet, and I. W. Jones. Explaining metastability in real synchronizers. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 59–67, May 2018.

[17] J. J. Reiher. Synchronizer analysis and design tool: an application to automatic differentiation. Master's thesis, University of British Columbia, 2020.

[18] S. M. Rump. *INTLAB — INTerval LABoratory*, pages 77–104. Springer Netherlands, Dordrecht, 1999.

[19] H. Vogt, M. Hendrix, and P. Nenzi. *ngSPICE - version 30.* 2019.

[20] L. Wei, O. Mysore, and D. Antoniadis. Virtual-source-based self-consistent current and charge FET models: From ballistic to drift-diffusion velocity-saturation operation. *IEEE Transactions on Electron Devices*, 59(5):1263–1271, May 2012.

[21] C. Yan. *Reachability Analysis Based Circuit-Level Formal Verification.* PhD thesis, The University of British Columbia, 2011.

[22] S. Yang and M. Greenstreet. Computing synchronizer failure probabilities. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

[23] S. Yang and M. R. Greenstreet. Simulating improbable events. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 154–157, New York, NY, USA, 2007. ACM.

[24] J. Zhou, D. Kinniment, G. Russell, and A. Yakovlev. A robust synchronizer. volume 2006, pages 2 pp.–, 04 2006.