

CMC Compact Model QA Specification



Revision History

Revision	Date	Authors	Comments
1.0	04/13/2006	C. C. McAndrew	Initial Release
1.1	05/08/2006	C. C. McAndrew	Verilog-A file specification added to Table 4.

Review History

Revision	Date	Reviewers

Table of Contents

1	Introduction	3
2	Preamble	3
3	Test Specification Requirements	4
	Table 1: Variants defined for testing.	5
4	Test Specification Definition and Test and Program Directory Structure	5
	Table 2: QA test file and directory structure and contents.	6
	Table 3: Header (test independent) keywords and their meanings.....	7
	Table 4: Test specific keywords and their meanings.	8
5	Test Result Generation	8
6	Reference Test Results	9
7	Procedure for Comparing Test Results	9
8	Example Code	10
Appendix 1.	Directory and File Hierarchy	11
Appendix 2.	Example qaSpec File for bsim3v3	12
Appendix 3.	Example Makefile for Running the Tests	14
Appendix 4.	Usage Message from the runQaTests.pl Program	19
Appendix 5.	Subroutines in perl Modules	20

1 Introduction

The Compact Model Council (CMC) is a body that operates under the auspices of the General Electrical Industries Association (GEIA) to standardize semiconductor device compact models. An important aspect of a model is proper quality assurance (QA), both to verify the correctness of the definition of a model and to verify the correctness of implementations of a model in circuit simulators. This document defines a process for compact model QA, including test requirements, test specification definition, provision of reference test results, and criteria for comparison to these test results. The QA procedure is intended to help verify the correctness of a compact model definition, by a model developer, and the correctness of a compact model implementation, by a simulator developer.

2 Preamble

Verification of the correctness of the definition of a compact model and the correctness of implementations of a compact model in a simulator are areas of compact modeling that need to be improved. Correctness of the physics embodied within a model and the accuracy with which a model can fit measured device characteristics are subjective, and are best done through human analysis (peer review). However, many aspects of a model definition and implementation are not subjective, yet have historically been the source of many problems with model definition and/or implementation.

At the model definition level, specification of multiplicity (m -factor) scaling, symmetric pin swapping (e.g. source and drain reversal for a MOSFET), and device polarity type (n - or p -polarity) have been the source of many errors in model definition and model implementations. This is because often the core model equations are developed and simulated primarily on a single device ($m=1$), of one device polarity type (n -type), and with symmetric pins biased only in one configuration ($V_{ds} \geq 0$).

At the model implementation level, because of the lack of provision of complete test specifications and results along with a model, there has been no standard method available to verify the implementation of a model in all simulators. For example, although basic $I(V)$ characteristics may be correct, more detailed parts of a device model, such as noise, have historically proven to give different results between different implementations.

To help model developers, model implementers, and model users, the CMC has defined a process for the testing of compact models. This document defines that process.

Defining and providing verified reference results for an exhaustive set of tests for a model is a significant amount of work. It is not expected that all existing models will have such a set of tests and test results generated for them. It is expected that models being developed will have the tests and test results generated as a standard part of the model development process. The incremental effort involved in doing this is small and, in fact, can reduce overall model development time as it provides a structure for verification of implementation of an incremental addition to a model. The physical correctness and accuracy of this model enhancement will need to be verified against data anyway, and the expectation is that the parameter sets and simulation results used for verification of the model enhancement would then form the basis of the test for this specific feature of the model.

It is difficult, if not impossible, to generate tests that provide 100% code coverage, especially in terms of every possible combination of conditional code within a model. However, the expectation is that, over time, tests should be defined that exercise all aspects of a model and if there are different

sections of code that are executed based on various parameter or bias values, then tests that exercise each section of code should be included as part of the standard QA test specification.

The expectation is every model will come with a set of test specifications and reference results. The reference results must be manually checked and verified to give the expected behavior. The test specifications will be in the form of a file that defines the tests in the format defined below and used in the example of Appendix 2. The test procedure is that the tests from the test specification file will be run on a specific implementation of a model, and that the test results will be generated and written to a file in a standard format. The most efficient and reliable way to do this is using automated scripts that parse the test specification file, set up and run the appropriate netlists for a specific simulator in which the implementation is being tested, and collate the results and write them to a file. It is not possible to provide such a test script for every simulator, however examples that indicate how this can be done are provided, for a variety of simulators and netlist formats. A directory and file naming convention, to allow a consistent structure for the results (both reference and those generated for one specific implementation verification) are defined. The results from running of the tests are then compared against the reference results. Scripts to do this comparison, with tolerances, are provided. Because of convergence criteria within simulators and different precision in which results are printed, a decision about whether two implementations are “the same” cannot be made based on an exact comparison of the results. The toleranced comparison is intended to provide a reasonable decision on whether two sets of results are the same (having to manually inspect hundreds or thousands of numbers to personally make such a decision is neither a pleasant nor an easy task). Automated checking that numbers are the “same” is not easy, and it is expected that there will be times that manual checking will be required, and that the script used for comparison will be continually improved over time.

For simplicity, under UNIX a Makefile can be set up to automatically run the tests, compare the tests results, and generate a report that allows a simple pass or fail evaluation.

Note: all numbers specified for the tests, including for instance and model parameters, must be in numerical format (including scientific notation) and must not use engineering unit qualifiers, because the latter differ between different simulators.

3 Test Specification Requirements

An individual test specification should exist for each “sub-model” of a model. For example, for a MOSFET, the gate current model is one specific “sub-model.” A set of parameters and reference test results should be provided for each sub-model, which primarily exercise that aspect of a model.

Because there can be interactions between sub-models, a test specification and test results should also be provided for the case where all sub-model effects are active. Given the combinatorial nature of inclusion of an enumerated list of separate effects, it is likely impractical to separately test all possible combinations. But it is expected that there should at least be an individual test for each sub-model effect and a test where all sub-model effects are activated. If there are any specific combinations that are expected to be likely to lead to problems with model definition or implementation, then separate tests should be provided for these combinations of sub-model effects.

Tests should be defined for DC, AC, and noise simulations, over a variety of biases, frequencies, device geometries, and temperatures.

For now, tests are not defined for transient analyses. This is because different time-step control algorithms in different simulators can generate results at different time points. Comparison of two

sets of transient simulations to determine if they are “the same” is thus difficult to do, and often is done based on visual inspection of plots of transient results.

If there are components of a model where alternative blocks of code are executed based on the settings of switches or the values of parameters, or also bias, geometry, or temperature, then parameter sets and/or bias/geometry/temperature values should be specified for testing so that the different blocks of code are tested. The goal should be 100% code coverage in tests, although as noted above this may not always be realistic to achieve. To keep the top level test directory uncluttered specific parameter sets used for testing should be stored in a sub-directory.

If a model has terminals that should be symmetric, such as source and drain for a MOSFET or JFET, then all tests should also be applied with biases to these terminals flipped.

If a model is applicable to different polarity devices, then all tests should be applied to both *n*- and *p*-polarity devices.

If a model supports the instance multiplicity parameter *m* then it should be tested for *m*=1 and at least one other value *m*>1.

There are efficient ways to set up netlists so that these last 3 items are tested automatically; this is described in Appendix 5. The different implementation aspects of a model, such as polarity, symmetric pin flipping, shrink, scale and *m*, are termed variants. The possible variants, used as identifiers in results file names are:

standard	no variation
noFlip_P	polarity changed from N-type (assumed reference) to P-type
flip_N	symmetric terminals flipped
flip_P	polarity changed from N-type to P-type and symmetric terminals flipped
shrink	shrink factor applied to instance parameters (%)
scale	scale factor applied to instance parameters
m	multiplicity factor applied to the device instance

Table 1: Variants defined for testing.

The test procedure is constructed so that separate tests for all of these variants do not have to be defined, there is a method for specifying which variants should be tested and then the QA procedure automatically tests these variants.

The conversion from instance parameter units of length to simulator units of length (the latter is usually meters), is multiplication by $scale*(1-shrink/100)$.

4 Test Specification Definition and Test and Program Directory Structure

Appendix 1 shows the directory and file hierarchy. The test definitions for a model (referred to here as “model”) are expected to be defined in a QA specification file called `qaSpec` at the top level directory for that model, and there should be a `Makefile` that runs the tests defined in the `qaSpec` file. For consistency, it is recommended that a separate directory `model` be set up for each model, that the perl programs and modules be stored in a directory `lib` at the same level as the `model` directories (and be added to your UNIX `path` environment variable). The program

runQaTests.pl is the script that runs the tests and compares the results (and is in the lib directory).

The structure for the files and results directories (all under the directory model, see Appendix 1 for a visual depiction) is:

Makefile	a file for the make program that runs the tests
qaSpec	the file with information defining the model and test information
parameters/	directory where model parameter sets are located
reference/	directory where reference results are located
reference/test.standard	reference results for the standard test variant of each test
results/	top level directory for results
results/simulator/	directory for test results for each simulator
results/simulator/vers/	directory for version vers of the simulator
results/simulator/vers/OS/	directory for each computer architecture and operating system
results/simulator/vers/OS/test.var	test results for variant var of each test

Table 2: QA test file and directory structure and contents.

The directory hierarchy includes identification of different versions of simulators and different computer platforms, as they can give different results and as each of these really does need to be tested independently. The OS identifier is formed from computer architecture, operating system name, and operating system version, joined with underscores. On a UNIX system these are determined from the *uname* command with the *-p*, *-s*, and *-r* flags, respectively. On a non-UNIX machine these are determined from the perl *Config* module. The subroutine *platform* in the *lib/modelQaTestRoutines.pm* file generates the appropriate directory name.

The format of the test specification file is as follows. The file consists of two sections, a header section with general information that is relevant to all tests, and a section that defines specific tests. The format is based on initial keywords in a line. Appendix 2 gives an example. The keyword *test* defines the start of information for that test, and ends the specification of the general information. If a keyword can be specified in both a general and a test specific block, the specification in the general information is the default for all tests, and a test specific specification overrides the global specification for that single test.

Comments are in the C++ style, anything from *//* to the end of a line is stripped out. Conditional statements can be included via *`ifdef`* *`else`* *`end`* constructs, and variables can be defined and undefined via *`define`* and *`undef`* directives. These may be placed anywhere in a QA spec file, and may be nested. The name of the simulator being tested is automatically set as a defined variable, so conditionals based on the simulator name included in QA specification files are automatically processed.

Qualifiers for some keywords can be lists; these can be either comma or space delimited.

The following can be specified in the general information section:

Keyword	Qualifiers	Description
[np]TypeSelectionArguments	argList	model selection specification (used to build the initial portion of a .model card, so includes parameters like level, version, type)
scaleParameters	shrink, scale, m	parameters that affect scaling of a device, shrink and scale affect instance parameters, m is device multiplicity; specifying a qualifier triggers testing of that variant
keyLetter	[a-z]	for simulators that require it the first letter of the instance line appropriate for the type of device model being tested, e.g. m for MOSFET
pins	p1 p2 ...	list of pin (terminal) names for the device
linearScale	iParList	list of instance parameters that have the units of dimension, e.g. l and w
areaScale	iParList	list of instance parameters that have the units of square dimension, e.g. as and ad
temperature	valueList	list of temperatures (°C) at which each test will be run
checkPolarity	[y n]	flag to switch on checking of polarity flipping (both polarity model selections must also be specified for this to be active)
symmetricPins	p1 p2	define pins for which a model should be symmetric

Table 3: Header (test independent) keywords and their meanings.

If both nTypeSelectionArguments and pTypeSelectionArguments are specified, and the symmetricPins keyword is also specified, then all four combinations of model polarity type and symmetric pin swapping are tested.

The following can be specified in the test specific section:

Keyword	Qualifiers	Description
test	name	defines the name for the test, results files are name.var where var is each variant that is exercised
temperature	valueList	list of temperatures (°C) at which each test will be run
biases	V(p1)=val V(p2)=val ...	define the voltage biases for each pin
biasList	V(pi)=val1, val2, ...	define a list of bias values for one pin, here pin pi
biasSweep	V(pk)=start, stop, step	define a bias sweep for one pin, here pin pk
freq	[lin dec oct] pts fMin fMax	frequency specification, for AC

		and noise tests, follows SPICE convention (Note: for <code>dec</code> and <code>oct pts</code> is the number of points per decade or octave, for <code>lin</code> it is the number of points)
<code>outputs</code>	<code>I(p1), I(p2), ...</code>	do a DC test, simulate and store currents in the specified terminals
<code>outputs</code>	<code>G(pi, pk), C(pj, pk), ...</code>	do an AC test, simulate and store the listed conductances and capacitance coefficients
<code>outputs</code>	<code>N(p1)</code>	do a noise test, simulate and store the noise current in the specified terminal (only one is allowed)
<code>instanceParameters</code>	<code>iPar1=val1 iPar2=val2 ...</code>	list of name=value pairs for instance parameters for the test
<code>modelParameters</code>	<code>[mPar=val parameterFile]</code>	model parameters for the model for the test, can be lists on name=value pairs, or lists of file names that contain instance parameters
<code>verilogaFile</code>	<code>filename</code>	name of the file that contains the Verilog-A definition of the model

Table 4: Test specific keywords and their meanings.

For some keywords, e.g. `scaleParameters` or `modelParameters`, multiple qualifiers for keywords are acceptable, all are used, and if there are multiple occurrences of these keywords in one test all qualifiers specified for all occurrences are used. For other keywords, only one occurrence of a keyword is expected; for multiple occurrences the last one in order in the file is used.

5 Test Result Generation

Because different simulators have different analysis capabilities, different netlists and analysis specifications and output formats, it is not possible to provide standard specifications of, or scripts to implement, how tests will be exercised for a specific simulator. The test types defined (DC, AC, and noise) are of sufficiently limited scope, and the test specifications are sufficiently simple, that it is anticipated that it is not too difficult to implement programs that will automatically run the specified tests and generate the required test results.

Example scripts for test results generation are available for some simulators (Spice3f5, Hspice©, Spectre©, and ADS©); if a model implementation is to be verified in another simulator appropriate routines that implement the tests must be provided. Appendix 5 gives some details.

The overall generation of test results should be controlled by a Makefile; Appendix 3 gives an example. This file should control the overall generation and comparison of test results. A script, `runQaTests.pl`, is provided that actually runs all of the tests; the Makefile runs this program. The script can also determine the computer architecture and operating system version, analyze the QA spec file and list the tests and variants that are defined, and determine which version of a simulator is being used. Appendix 4 lists the help message from `runQaTests.pl`, which details the possible

calling options. The Makefile uses different calls to this script to control automated running of the tests and variants.

The `runQaTests.pl` script automatically stores test results according to the directory hierarchy and file naming conventions defined above.

6 Reference Test Results

Reference test results, that need to be manually verified, are expected to be provided as the “golden” standard against which model implementations are compared. These are to be stored in the directory `reference`, for each test. Only results for the standard variant should be provided; test results for other variants are expected to be manipulated (by the test routines, via the mechanism described in Appendix 5) so that they provide results identical to the standard. The automated test script compares results of the standard variant simulations to the reference test results, and compares results of simulations for other variants to those of the standard results (not the reference results). This is because the variant tests are designed to, as much as is possible, give results that exactly match those of the standard variant (i.e. there should not be slight differences in the last digit of precision printed by a simulator that depend on polarity or any other variant property). There may be slight differences for results generated from one platform or simulator because of different convergence criteria, different formats for printing of results, different compilers or compiler options, etc. These differences will only show up in the information message printed when comparing the standard variant simulations to the reference results, and not when other variants are compared to the standard for one simulator, version, architecture, operating system, etc. This makes the automated regression test output cleaner to look at and easier to interpret.

7 Procedure for Comparing Test Results

Comparing two sets of simulation results to check if they are the same or different is not a simple task. Differences in convergence algorithms, compilers or compiler options used to generate the simulator executable, and printing formats can lead to differences in results between different simulators that are not indicative of differences in model implementations between the simulators (i.e. of errors in one or more implementations). Also, at low current levels, differences in how things like g_{\min} are handled can lead to differences in simulation results not really tied to problems in the implementation of the core model. These differences can still be a sufficient criterion to reject a particular implementation as not matching the reference standard.

Aside: to prevent “false positives” in testing differences in model implementations in different simulators, it can be useful to define the tests so that they do not include evaluation when the bias across a device is zero, where simulation differences can be small in an absolute sense but large in a relative sense when the target result is zero current. Simulation of bias sweeps through zero bias are useful in evaluating model symmetry; specific automated tests for DC and AC symmetry may be added in future, but for now should be part of the evaluation and verification process during model development.

The script `compareSimulationResults.pl` is provided to help, and standardize, numerical comparison of results. This script compares results with relative and absolute comparison criteria tied to the analysis type (checking for differences in noise currents of order 10^{-22} is very different from checking for differences in DC currents of order 1mA). This script returns strings that qualitatively evaluate the comparison between results. Possible comparison returns are: failure if there are

different numbers of simulation and reference results (likely a simulation failure); difference outside the prescribed tolerances; match within the prescribed tolerances; and an exact match.

The Makefile, test program, and comparison program are constructed so that when they run coherently they generate a nicely formatted output that is easy to visually scan to evaluate success or failure of the QA test run.

8 Example Code

The simplest way to set up the appropriate test files and directories is to copy an existing example. An example, set up for the BSIM3v3 model, is available at

<http://www.eigroup.org/cmc/downloads/default.htm>

and includes a test specification file, a make file to run the tests, complete reference results, and all of the programs (in perl) to run the tests and compare the results. All examples in this document are from this set of reference QA procedures.

The examples provided run simulations for: Spice3f5, Hspice©, Spectre©, and ADS©.

Appendix 1. Directory and File Hierarchy

```
lib/
  runQaTests.pl
  modelQaTestRoutines.pm
  compareSimulationResults.pl
  spice.pm
  otherSimulator.pm
model1/
  Makefile
  qaSpec          <- this is the test specification file
  parameters/
    parameterFile1
    parameterFile2
    ...
  reference/
    testName1.standard
    testName2.standard
    ...
  results/
    simulator1/
      version1/
        OS1/
          testName1.standard
          testName1.variant1
          testName1.variant2
          ...
          testName2.standard
          testName2.variant1
          testName2.variant2
          ...
        simulatorJ/
          versionK/
            OSL/
              testNameM.variantN
model2/
  Makefile
  qaSpec
  parameters/
```

Appendix 2. Example qaSpec File for bsim3v3

```
//  
// Example test specification for bsim3v3 (version 3.2.4)  
//  
  
//  
// Simulator specific information  
// These arguments are added to the model card  
// specification to invoke the desired model in  
// different simulators (which can have different  
// names or levels for the same model) and to switch  
// between nType and pType polarities.  
// It is assumed that there are no polarity specific  
// parameters.  
//  
  
`ifdef spice  
nTypeSelectionArguments      nmos level=7 version=3.2.4  
pTypeSelectionArguments      pmos level=7 version=3.2.4  
`endif  
`ifdef hspice  
nTypeSelectionArguments      nmos level=53 version=3.24  
pTypeSelectionArguments      pmos level=53 version=3.24  
scaleParameters             scale  
`endif  
`ifdef spectre  
nTypeSelectionArguments      bsim3v3 type=n version=3.24  
pTypeSelectionArguments      bsim3v3 type=p version=3.24  
scaleParameters             scale  
`endif  
`ifdef mica  
nTypeSelectionArguments      nmos level=9 version=3.2.4  
pTypeSelectionArguments      pmos level=9 version=3.2.4  
scaleParameters             scale shrink  
`endif  
`ifdef ads  
nTypeSelectionArguments      MOSFET Idsmod=8 NMOS=yes Version=3.24  
pTypeSelectionArguments      MOSFET Idsmod=8 PMOS=yes Version=3.24  
`endif  
  
//  
// General test-independent information  
//  
  
keyLetter                    m  
pins                          d g s b  
linearScale                   w l ps pd  
areaScale                     as ad  
temperature                    27 -50 150  
checkPolarity                 yes  
symmetricPins                 d s  
scaleParameters               m
```

```

//
// Specific tests
//

test          dcSweep
biases        V(s)=0 V(b)=0
biasList      V(g)=0.4,0.6,0.8,1.0,1.2
biasSweep     V(d)=0.1,1.2,0.1
outputs       I(d)
instanceParameters w=10.0e-6 l=1.0e-6
modelParameters parameters/nmosParameters

test          acVd
temperature   27 150
biases        V(s)=0 V(b)=0 V(g)=1.2
biasSweep     V(d)=0.1,1.2,0.1
outputs       G(d,g) G(d,d) C(g,s) C(g,d)
instanceParameters w=10.0e-6 l=1.0e-6
modelParameters parameters/nmosParameters

test          acFreq
temperature   27
biases        V(s)=0 V(b)=0 V(d)=1.2 V(g)=1.2
freq          dec 10 1e3 1e9
outputs       C(g,g) C(g,s) C(g,d)
instanceParameters w=10.0e-6 l=1.0e-6
modelParameters parameters/nmosParameters

```

Appendix 3. Example Makefile for Running the Tests

```
#
# Example Makefile to run tests and check results.
#
# This is an example makefile for running QA tests on a
# model and then checking the simulated results against
# reference results. A separate target is defined for each
# variant of the model. The program runQaTests.pl runs the
# tests, and that program expects a perl module SIMULATOR.pm
# to be provided for each simulator that is tested.
# Examples of these are provided.
#

qaSpecFile           =      qaSpec
qaResultsDirectory  =      results
testProgramName     =      runQaTests.pl

help:
    @echo "" ; \
    echo "Valid targets are:" ; \
    echo "" ; \
    echo "all          run tests and compare results for all simulators" ; \
    echo "" ; \
    echo "spectre      run tests and compare results spectre" ; \
    echo "hspice        run tests and compare results hspice" ; \
    echo "ads           run tests and compare results ads" ; \
    echo "spice         run tests and compare results spice" ; \
    echo "" ; \
    echo "clean         remove all previously generated simulation results"; \
    echo "" ; \
    echo "NOTE: if test results exist they are not resimulated" ; \
    echo "NOTE: to force resimulation run \"make clean\" first" ; \
    echo ""

all: spectre hspice ads spice

#####
##### spectre tests
#####

spectre:
    @-echo "" ; \
    localPlatform=`$(testProgramName) -platform` ; \
    localVersion=`$(testProgramName) -sv -s spectre $(qaSpecFile)` ; \
    localVersionAndPlatform=${localVersion}._.${localPlatform} ; \
    echo "*****"; \
    echo "***** $(qaSpecFile) tests for spectre"; \
    echo "***** (for version $$localVersion on platform $$localPlatform)"; \
    echo "*****"; \
    for test in `$(testProgramName) -lt -s spectre $(qaSpecFile)` ; \
    do \
        echo "" ; \
        echo "***** Checking test (spectre): $$test" ; \
        for var in `$(testProgramName) -lv -s spectre $(qaSpecFile)` ; \
        do \
            $(testProgramName) -s spectre -r -t $$test -var $$var $(qaSpecFile) ; \
```

```

done ; \
done ; \
for version in `ls -Cl $(qaResultsDirectory)/spectre` ; \
do \
  for platform in `ls -Cl $(qaResultsDirectory)/spectre/$$version` ; \
  do \
    versionAndPlatform=$$version._.$$platform ; \
    if [ $$versionAndPlatform = $$localVersionAndPlatform ] ; \
    then \
      break ; \
    fi ; \
    echo "" ; \
    echo "*****"; \
    echo "***** Comparing existing $(qaSpecFile) tests for spectre"; \
    echo "***** (for version $$version on platform $$platform)"; \
    echo "*****"; \
    for test in `$(testProgramName) -lt -s spectre $(qaSpecFile)` ; \
    do \
      echo ""; \
      echo "***** Checking test (spectre): $$test" ; \
      for var in `$(testProgramName) -lv -s spectre $(qaSpecFile)` ; \
      do \
        $(testProgramName) -c $$version $$platform \
          -s spectre -t $$test -var $$var $(qaSpecFile) ; \
      done ; \
    done ; \
  done ; \
done ; \
done

```

```

#####
##### hspice tests
#####

```

hspice:

```

@-echo "" ; \
localPlatform=`$(testProgramName) -platform` ; \
localVersion=`$(testProgramName) -sv -s hspice $(qaSpecFile)` ; \
localVersionAndPlatform=$$localVersion._.$$localPlatform ; \
echo "*****"; \
echo "***** $(qaSpecFile) tests for hspice"; \
echo "***** (for version $$localVersion on platform $$localPlatform)"; \
echo "*****"; \
for test in `$(testProgramName) -lt -s hspice $(qaSpecFile)` ; \
do \
  echo ""; \
  echo "***** Checking test (hspice): $$test" ; \
  for var in `$(testProgramName) -lv -s hspice $(qaSpecFile)` ; \
  do \
    $(testProgramName) -s hspice -r -t $$test -var $$var $(qaSpecFile) ; \
  done ; \
done ; \
for version in `ls -Cl $(qaResultsDirectory)/hspice` ; \
do \
  for platform in `ls -Cl $(qaResultsDirectory)/hspice/$$version` ; \
  do \
    versionAndPlatform=$$version._.$$platform ; \
    if [ $$versionAndPlatform = $$localVersionAndPlatform ] ; \
    then \

```

```

        break ; \
    fi ; \
    echo "" ; \
    echo "*****"; \
    echo "***** Comparing existing $(qaSpecFile) tests for hspice"; \
    echo "***** (for version $$version on platform $$platform)"; \
    echo "*****"; \
    for test in `$(testProgramName) -lt -s hspice $(qaSpecFile)` ; \
    do \
        echo ""; \
        echo "***** Checking test (hspice): $$test" ; \
        for var in `$(testProgramName) -lv -s hspice $(qaSpecFile)` ; \
        do \
            $(testProgramName) -c $$version $$platform \
                -s hspice -t $$test -var $$var $(qaSpecFile) ; \
        done ; \
    done ; \
done ; \
done
#####
##### ads tests
#####

ads:
@-echo "" ; \
localPlatform=`$(testProgramName) -platform` ; \
localVersion=`$(testProgramName) -sv -s ads $(qaSpecFile)` ; \
localVersionAndPlatform=$$localVersion._.$$localPlatform ; \
echo "*****"; \
echo "***** $(qaSpecFile) tests for ads"; \
echo "***** (for version $$localVersion on platform $$localPlatform)"; \
echo "*****"; \
for test in `$(testProgramName) -lt -s ads $(qaSpecFile)` ; \
do \
    echo ""; \
    echo "***** Checking test (ads): $$test" ; \
    for var in `$(testProgramName) -lv -s ads $(qaSpecFile)` ; \
    do \
        $(testProgramName) -s ads -r -t $$test -var $$var $(qaSpecFile) ; \
    done ; \
done ; \
for version in `ls -Cl $(qaResultsDirectory)/ads` ; \
do \
    for platform in `ls -Cl $(qaResultsDirectory)/ads/$$version` ; \
    do \
        versionAndPlatform=$$version._.$$platform ; \
        if [ $$versionAndPlatform = $$localVersionAndPlatform ] ; \
        then \
            break ; \
        fi ; \
        echo "" ; \
        echo "*****"; \
        echo "***** Comparing existing $(qaSpecFile) tests for ads"; \
        echo "***** (for version $$version on platform $$platform)"; \
        echo "*****"; \
        for test in `$(testProgramName) -lt -s ads $(qaSpecFile)` ; \
        do \
            echo ""; \

```

```

        echo "***** Checking test (ads): $$test" ; \
        for var in `$(testProgramName) -lv -s ads $(qaSpecFile)` ; \
        do \
            $(testProgramName) -c $$version $$platform \
                -s ads -t $$test -var $$var $(qaSpecFile) ; \
        done ; \
    done ; \
done ; \
done

#####
##### spice tests
#####

spice:
@-echo "" ; \
localPlatform=`$(testProgramName) -platform` ; \
localVersion=`$(testProgramName) -sv -s spice $(qaSpecFile)` ; \
localVersionAndPlatform=$$localVersion._.$$localPlatform ; \
echo "*****" ; \
echo "***** $(qaSpecFile) tests for spice" ; \
echo "***** (for version $$localVersion on platform $$localPlatform)" ; \
echo "*****" ; \
for test in `$(testProgramName) -lt -s spice $(qaSpecFile)` ; \
do \
    echo "" ; \
    echo "***** Checking test (spice): $$test" ; \
    for var in `$(testProgramName) -lv -s spice $(qaSpecFile)` ; \
    do \
        $(testProgramName) -s spice -r -t $$test -var $$var $(qaSpecFile) ; \
    done ; \
done ; \
for version in `ls -Cl $(qaResultsDirectory)/spice` ; \
do \
    for platform in `ls -Cl $(qaResultsDirectory)/spice/$$version` ; \
    do \
        versionAndPlatform=$$version._.$$platform ; \
        if [ $$versionAndPlatform = $$localVersionAndPlatform ] ; \
        then \
            break ; \
        fi ; \
        echo "" ; \
        echo "*****" ; \
        echo "***** Comparing existing $(qaSpecFile) tests for spice" ; \
        echo "***** (for version $$version on platform $$platform)" ; \
        echo "*****" ; \
        for test in `$(testProgramName) -lt -s spice $(qaSpecFile)` ; \
        do \
            echo "" ; \
            echo "***** Checking test (spice): $$test" ; \
            for var in `$(testProgramName) -lv -s spice $(qaSpecFile)` ; \
            do \
                $(testProgramName) -c $$version $$platform \
                    -s spice -t $$test -var $$var $(qaSpecFile) ; \
            done ; \
        done ; \
    done ; \
done
done

```

clean:

```
@/bin/rm -rf $(qaResultsDirectory)/spectre spectreCkt* *ahdlcmi  
@/bin/rm -rf $(qaResultsDirectory)/hspice hspiceCkt*  
@/bin/rm -rf $(qaResultsDirectory)/ads adsCkt* spectra.raw  
@/bin/rm -rf $(qaResultsDirectory)/spice spiceCkt* b3v3check.log
```

Appendix 4. Usage Message from the runQaTests.pl Program

runQaTests.pl: run model QA tests

Usage: runQaTests.pl [options] -s simulatorName qaSpecificationFile

Files:

qaSpecificationFile	file with specifications for QA tests
simulatorName	name of simulator to be tested

Options:

-c version platform	do not try to simulate, only compare results for version and platform
-d	debug mode (leave intermediate files around)
-h	print this help message
-i	print info on file formats and structure
-l	list tests and variants that are defined
-lt	list tests that are defined
-lv	list test variants that are defined
-platform	prints the hardware platform and operating system version
-p	plot results (limited, only standard test variant)
-P	plot results (complete, for all test variants)
-r	re-use previously simulated results if they exist (default is to resimulate, even if results exist)
-sv	prints the simulator version being run
-t TEST	only run test TEST (can be a comma delimited list)
-var VAR	only run variant VAR (can be a comma delimited list)
-v	verbose mode
-V	really verbose mode, print out each difference detected

The `-lt` and `-lv` flags are used by the Makefile to determine which tests and variants to loop over.

The `-sv` and `-platform` flags are used by the Makefile to determine the simulator version and computer architecture and operating system information, and these are used as hierarchy levels in the directory structure that stores the test results.

Appendix 5. Subroutines in perl Modules

Note: the `runQaTests.pl` program expects to find the perl modules that it requires and the `compareSimulationResults.pl` program in the directory `../lib` relative to where it is run. This is the structure defined in Appendix 1.

The following subroutines are in the `modelQaTestRoutines.pm` perl module:

<code>processSetup</code>	processes test independent information, sets up variables used in the simulation routines
<code>processTestSpec</code>	processes the specific information for each test, sets up variables used in the simulation routines
<code>readQaSpecFile</code>	parses the <code>qaSpec</code> file and sets up generic and test specific information
<code>processIfDefs</code>	Processes conditionals in the <code>qaSpec</code> file
<code>unScale</code>	converts from engineering/SPICE number notation to pure numbers
<code>platform</code>	Returns computer architecture and operating system information

The perl module provided for each simulator must have the following subroutines defined:

<code>version</code>	returns the simulator version
<code>runDcBiasTest</code>	runs and returns the results of specified DC tests
<code>runAcBiasTest</code>	runs and returns the results of specified AC tests
<code>runNoiseTest</code>	runs and returns the results of specified noise tests
<code>generateCommonNetlistInfo</code>	(recommended) sets up the model in the simulator netlist

The `generateCommonNetlistInfo` subroutine is separated for convenience, and is not mandatory. However, the model set up should be the same for each test, handling the test variants (polarity, symmetric pin swapping, shrink, scale, multiplicity) needs to be done for every test type, and duplication of common code is undesirable, hence it is recommended that the test routines be structured so they call a common routine for generation of the model, as a sub-circuit because this greatly simplifies implementation of the test variants.

The sub-circuit operates as follows. It contains a single instance of the device, and uses unity gain voltage-controlled voltage sources (VCVSs) to drive the pins of the device based on the biases applied to the pins of the sub-circuit. Unity gain current-controlled current sources (CCCSs) are then used to mirror the currents in the voltage sources to the terminals of the sub-circuit.

To check results for polarity reversal (e.g. PMOS compared to NMOS), the sign of the gain of the VCVSs and CCCSs is flipped. Note that this makes the sign of PMOS voltages and currents the same as for NMOS. The reason for this is that some simulators print results in fixed width fields, so if one number has a negative sign in front and another does not, the numbers will be printed to different numbers of digits of precision. This complicates checking for an exact match of numbers. Note that for this test to work effectively, a model needs to behave identically for both polarity configurations, and parameters for which there are different defaults for different polarities need to be set. If a model has conditional code that depends on polarity, and parameters cannot be set so that this code is inactive, then there is no point applying this test. In this case, separate tests, with different reference results due to real differences in model evaluations, need to be defined for each polarity.

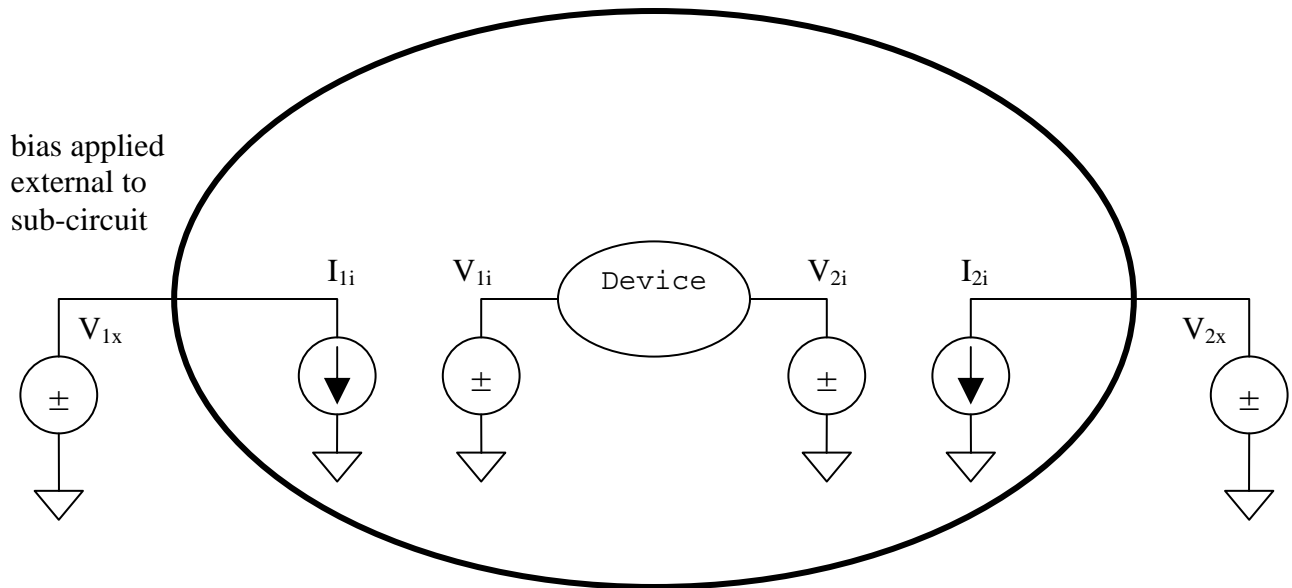
To check for symmetry pin reversal, the control pins for the VCVSs are interchanged, and similarly for the CCCSs.

To check for polarity flipping and symmetry pin reversal, both of the previous changes are made.

To check scale or shrink values, if they are supported, simulator options or variables that control these are set, and model instance parameters are scaled by the inverse of the scale or shrink factors (or the square of this if the instance parameter is a measure of area).

To check multiplicity, an m-factor is included for the device instance, and the CCCS gains are set to the inverse of the device instance m-factor. As with the polarity reversal, this guarantees that for simulators with tightly controlled output printing formats, the numbers should exactly match those of the standard simulation.

The following shows the sub-circuit that enables easy testing of variants.



In normal operation, $V_{1i}=+V_{1x}$, $I_{1i}=-I(V_{1i})$, $V_{2i}=+V_{2x}$, $I_{2i}=-I(V_{2i})$ (the negative sign for the current-controlled current sources is because of the polarity of the current in the voltage-controlled voltage sources internal to the sub-circuit).

For reversed polarity operation, $V_{1i}=-V_{1x}$, $I_{1i}=+I(V_{1i})$, $V_{2i}=-V_{2x}$, $I_{2i}=+I(V_{2i})$.

For terminal swapped operation, $V_{1i}=+V_{2x}$, $I_{1i}=-I(V_{2i})$, $V_{2i}=+V_{1x}$, $I_{2i}=-I(V_{1i})$.

For m-factor checking, the internal device instance inside the sub-circuit has the scale factor set to m, and then $V_{1i}=-V_{1x}$, $I_{1i}=+I(V_{1i})/m$, $V_{2i}=-V_{2x}$, $I_{2i}=+I(V_{2i})/m$. (This is modified for noise simulations where the noise current in A^2/Hz is being measured; the current sources are then scaled by $1/\sqrt{m}$ rather than by $1/m$).