Qucs

A Description

Verilog-AMS interface

Stefan Jahn
Hélène Parruitte

Introduction

Verilog-AMS is a hardware description language. It can be used to specify the analogue behaviour of compact device models. Usually these are C or C++ implementations in analogue simulators. The effort to implement modern compact models in C/C++ is quite high compared to the description in Verilog-AMS.

ADMS

The software ADMS (see `http://mot-adms.sourceforge.net`) allows Verilog-AMS descriptions to be translated into any other programming language. It generates a structured XML tree representing the compact device model description.

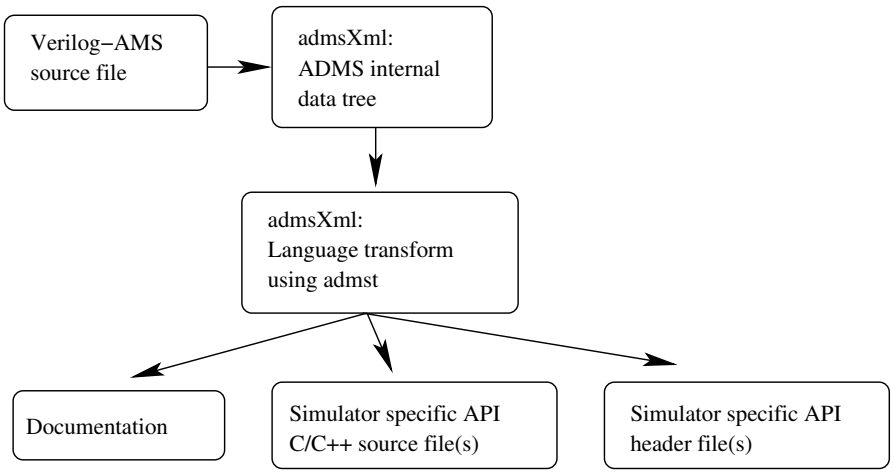

Figure 1: ADMS data flow

The internal XML tree is used to generate ready-to-compile C or C++ code which is specific to the simulators API. The code generator is able to produce

- evaluation of device equations (current and charge) including their derivatives,
- glue code for the simulator API,
- documentation and
- any other data described by the original Verilog-AMS input file.

XML admst scripts

The language transformation uses a language named **admst**. It is itself a XML description. The command line in order to run a transformation is

```
$ admsXml <device.va> -e <interface-1.xml> -e <interface-2.xml>
```

From version 0.0.11 Qucs comes with the following Verilog-AMS transformers

- `qucsMODULEcore.xml`
  creating the actual analogue simulator implementation
- `qucsMODULEdefs.xml`
  creating the parameter descriptions for the analogue simulator
- `qucsMODULEgui.xml`
  creating the implementation for the GUI integration
- `qucsVersion.xml`
  basic admst library
- `analogfunction.xml`
  creating analogue function code

In order to create **admst** scripts for a simulator it is necessary to understand both, the simulator specific API and how the ADMS data tree items – which are based on the Verilog-AMS source file describing the model – relate to the API.

The command lines for transforming a Verilog-AMS source file into the appropiate Qucs C++ source files are

```
$ admsXml <device.va> -e qucsVersion.xml -e qucsMODULEcore.xml
$ admsXml <device.va> -e qucsVersion.xml -e qucsMODULEgui.xml
$ admsXml <device.va> -e qucsVersion.xml -e qucsMODULEdefs.xml
$ admsXml <device.va> -e analogfunction.xml
```

each creating an appropriate `*.cpp` and `*.h` file.

The **admst** language is used to traverse the internal tree. The tree's root is defined by the Verilog-AMS module definition.

```
module device (node1, node2, ...)
  // module definitions and code
endmodule
```

Short introduction into the admst language syntax

Usually the **admst** language instructions are basically formed as

```
<admst:instruction argument=...>
  ...
</admst:instruction>
```

or

```
<admst:instruction argument=.../>
```

Any other text outside these instruction is output to the console or into an appropriate file. Some of the most important language construct are listed below.

- Traversing a list: The construct allows to traverse all children of the selected branch.

  ```
  <admst:for-each select="/module">
    ...
  </admst:for-each>
  ```

- Defining and using variables: Values from the data tree can be put into named and typed variables which are then accessible using the $ operator.

  ```
  <admst:value-of select="name">
  <admst:variable name="module" select="%s">
  ```

- Opening a file: Printed text (see below) is output into the given file.

  ```
  <admst:open file="$module.cpp">
    ...
  </admst:open>
  ```

- Output text: Special characters must be encoded (e.g. `"` → `&quot;` `<` → `<` `>` → `>`; and `\n` → `\\n`).

  ```
  <admst:text format="This is a text."/>
  ```

- Definition of a function: Functions in **admst** are called templates.

  ```
  <admst:template match="name_of_function">
    ...
  </admst:template>
  ```

- Running a function: Templates are applied to parts of the internal data tree.

```
<admst:apply-templates select="date_tree_root_for_function"
                       match="name_of_function"/>
```

- Comments:

  <!-- this is a comment -->

Analogue simulator script

The analogue simulator implementation consists of several parts. For each type of simulation appropriate functions must be implemented.

- DC simulation
  `module::initDC (void)`,
  `module::restartDC (void)` and
  `module::calcDC (void)`
- AC simulation
  `module::initAC (void)` and
  `module::calcAC (nr_double_t)`
- S-parameter simulation
  `module::initSP (void)` and
  `module::calcSP (nr_double_t)`
- Transient simulation
  `module::initTR (void)` and
  `module::calcTR (nr_double_t)`
- AC noise simulation
  `module::initNoiseAC (void)` and
  `module::calcNoiseAC (nr_double_t)`
- S-parameter noise simulation
  `module::initNoiseSP (void)` and
  `module::calcNoiseSP (nr_double_t)`
- Harmonic simulation
  `module::initHB (int)` and
  `module::calcHB (int)`

These functions go into the `module.core.cpp` file. An appropriate `module.core.h` header file is also created.

In case the Verilog-AMS source file contains analog functions in the module definition, e.g.

```
analog function real name;
  input x;
  real x;
  begin
    name = x * x;
  end
endfunction
```

the `analogfunction.xml` script produces the appropriate C/C++ and header files

- `module.analogfunction.cpp`
- `module.analogfunction.h`.

The ADMS language transformer is aware of how analogue simlators solve a network of linear and non-linear devices. The general Newton-Raphson algorithm for a DC simulation used in SPICE-like simulators as well as in Qucs can be expressed as

$$\begin{aligned} J^{(m)} \cdot V^{(m+1)} &= J^{(m)} \cdot V^{(m)} - f\left(V^{(m)}\right) \\ &= I_{lin}^{(m)} + I_{nl}^{(m)} - J_{nl}^{(m)} \cdot V^{(m)} \end{aligned} \tag{1}$$

whereas J denotes the Jacobian

$$J^{(m)} = \left.\frac{\partial f\left(V\right)}{\partial V}\right|_{V^{(m)}} \tag{2}$$

There are basically two types of contributions supported by ADMS: currents and charges. The appropiate Jacobian matrices are

$$J^{(m)} = J_I^{(m)} + J_Q^{(m)} = \underbrace{\left.\frac{\partial I\left(V\right)}{\partial V}\right|_{V^{(m)}}}_{\text{“static”}} + \underbrace{\left.\frac{\partial Q\left(V\right)}{\partial V}\right|_{V^{(m)}}}_{\text{“dynamic”}} = G + C \tag{3}$$

consisting of two real valued matrices G (conductance/transconductance matrix) and C (capacitance/transcapacitance matrix).

In the Verilog-AMS desciptions only the current and charge contributions are mentioned. The appropriate derivatives are meant to be automatically formed by the language translator ADMS.

```
I(ci,ei) <+ it;          // current contribution
I(si,ci) <+ ddt(Qjs); // charge contribution
```

The right-hand side of eq. (1) consists of the linear and non-linear currents of the network as well as the Jacobian matrix multiplied by the voltage vector. Since devices are usually uncorrelated both parts can be computed directly on a per device basis.

Current limitations of ADMS

The following section contains some simple examples demonstrating which Verilog-AMS statements can be successfully handled by ADMS and which not.

Example 1

There was a bug in the **admst** scripts. They failed to place the function calls of analogue functions correctly. This has been fixed.

```
module diode(a, c);
  inout a, c;
  electrical a, c;

  analog function real current;
    input is, v;
    begin
      current = is * (exp (v / 26e-3) - 1);
    end
  endfunction

  real Vd;
  real Id;

  analog begin
    Vd = V(a, c);
    Id = current (1e-15, Vd);
    I(a, c) <+ Id;
  end

endmodule
```

Example 2

It is not allowed to mix current and charge contributions in assignments. Mixing both emits wrong C/C++ code. It accounts the current to be a charge in this case.

```
module diode(a, c);
  inout a, c;
  electrical a, c;

  real Vd, Id, Is, Cp;

  analog begin
    Is = 1e-15;
    Cp = 1e-12;
    Vd = V(a, c);
    Id = Is * (exp (Vd / 26e-3) - 1);

    // not allowed in ADMS
    I(a, c) <+ Id + ddt(Cp*V(a,c));

    // allowed in ADMS
    I(a, c) <+ Id;
    I(a, c) <+ ddt(Cp*V(a,c));
  end

endmodule
```

Example 3

The following example is rejected by the **admst** scripts with this error message.

```
[info] admsXml-2.2.4 Oct 18 2006 19:50:46
[fatal] qucsVersion.xml:1497:admst:if[lhs]: missing node source/insource
```

An immediate potential on the right hand side is not allowed embedded in a function.

```
module diode(a, c);
  inout a, c;
  electrical a, c;

  real Id, Is;

  analog begin
    Is = 1e-15;
    Id = Is * (exp (V(a, c) / 26e-3) - 1);
```

```
    // not allowed in ADMS
    I(a, c) <+ Is * (exp (V(a, c) / 26e-3) - 1);

    // allowed in ADMS
    I(a, c) <+ Id;
  end

endmodule
```

Example 4

Potentials on the left-hand side of contribution assignments are not allowed. This kind of assignment emits wrong C/C++ code.

```
module diode(a, c);
  inout a, c;
  electrical a, c;

  real Id, Is;
  real Rp;

  analog begin
    Rp = 1e9;
    Is = 1e-15;
    Id = Is * (exp (V(a, c) / 26e-3) - 1);
    I(a, c) <+ Id;

    // allowed in ADMS
    I(a, c) <+ V(a, c) / Rp;

    // not allowed in ADMS
    V(a, c) <+ I(a, c) * Rp;
  end

endmodule
```

Code for the GUI integration

The **admst** script for the GUI creates files according to its API. Basically it produces the list of parameters as well their descriptions (if available) in a property

list. Additionally the symbol drawing code is emitted which should be adapted to the devices requirements.

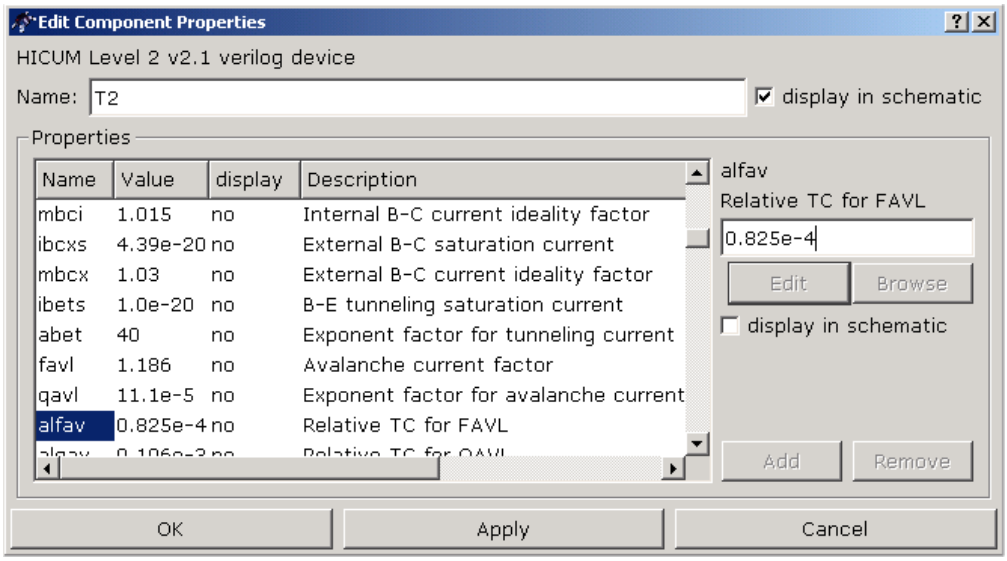

Figure 2: component property dialog in the GUI

In fig. 2 the component property dialog of an implemented device is shown. In the schematic in fig. 3 the symbol for the HICUM model is surrounded with a red circle. The GUI code is also responsible for the icons in the listview on the left hand side of the figure.

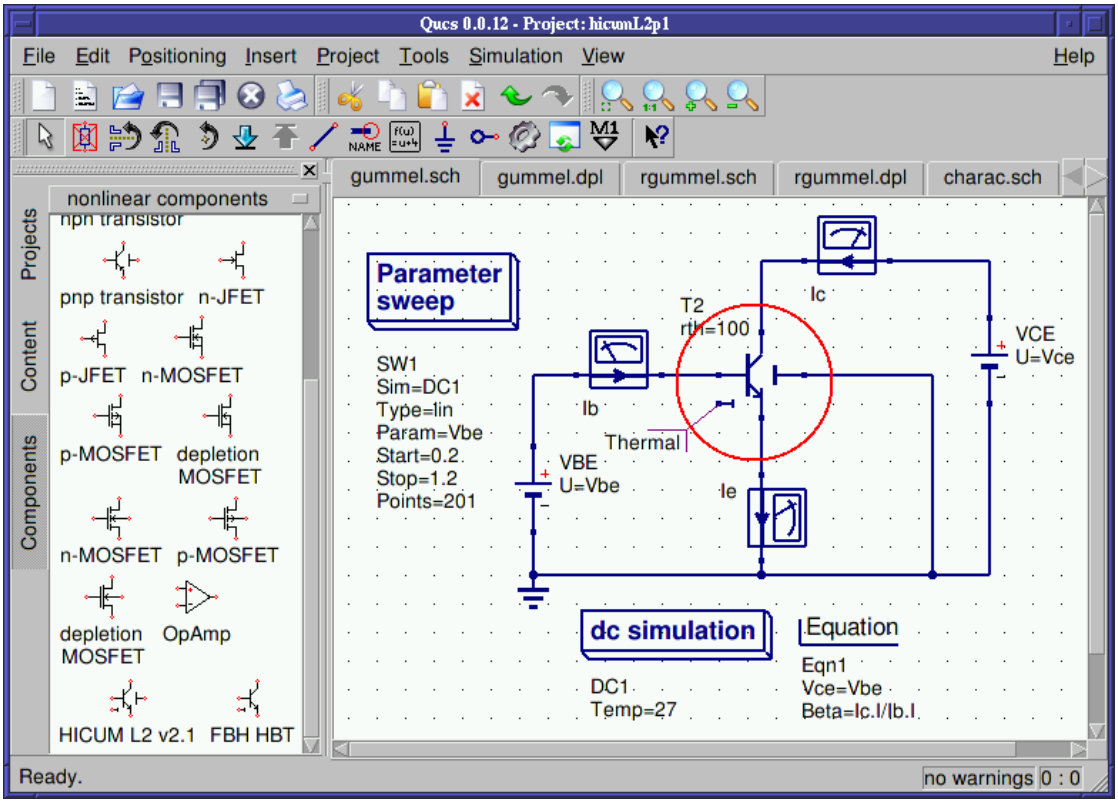

Figure 3: schematic with the HICUM transistor symbol

Adding Verilog-AMS devices to Qucs

The section gives an overview how to add a Verilog-AMS model to Qucs in a step-by-step manner. Be aware that the description is meant for advanced users partly familiar with the usual GNU/Linux build system and C/C++ programming.

The Verilog-AMS source file

The starting point of a new Verilog-AMS model in Qucs is the Verilog-AMS source file which must be aware of the discussed ADMS limitations. The example we are going to discuss is a simple diode model shown in the following listing contained in the **diode.va** file.

```
‘include ”disciplines.vams”
‘include ”constants.vams”

// ADMS specific definitions
```

```
`define attr(txt) (*txt*)

module diode(a, c);
  // device terminals
  inout a, c;
  electrical a, c;

  // internal node
  electrical ci;

  // model parameters
  parameter real Is   = 1E-15 from [0:1]
    `attr(info="saturation current");
  parameter real Cp   = 1E-12 from [0:1]
    `attr(info="parallel capacitance");
  parameter real Rs   = 1.0   from (0:inf)
    `attr(info="series resistance");
  parameter real Temp = 300   from (0:inf)
   `attr(info="temperature");

  real Vd, Id, fourkt, twoq, Qp;

  analog begin
    Vd = V(a, ci);
    Id = Is * (exp (Vd / 26e-3) - 1);
    Qp = Cp * Vd;

    I(a, ci) <+ Id;
    I(a, ci) <+ ddt(Qp);
    I(ci, c) <+ V(ci, c) / Rs;

  begin : noise
    fourkt  = 4.0 * `P_K * Temp;
    twoq    = 2.0 * `P_Q;
    I(ci, c) <+ white_noise(fourkt/Rs, "thermal");
    I(a, ci) <+ white_noise(twoq*Id, "shot");
  end // noise

  end // analog
```

endmodule

Integrating the model into the analogue simulator

The qucs-core tree of Qucs must be configured using the –enable-maintainer-mode option.

```
$ ./configure --enable-maintainer-mode --prefix=/tmp
```

The source trees of qucs (GUI) as well as qucs-core (simulator) are within the release tarballs available at `http://qucs.sourceforge.net/download.html`. During development of new Verilog-A models it is recommended to use CVS. Latest CVS trees of qucs and qucs-core can be obtained using the following command lines.

```
$ cvs -z3 -d:pserver:anonymous@qucs.cvs.sourceforge.net:/cvsroot/qucs \
  co qucs-core
$ cvs -z3 -d:pserver:anonymous@qucs.cvs.sourceforge.net:/cvsroot/qucs \
  co qucs
```

Also the software **adms** must be installed. It is available at `http://sourceforge.net/projects/mot-adms`.

The file **diode.va** must be copied into the source tree.

```
$ cp diode.va qucs-core/src/components/verilog/
```

In this directory (qucs-core/src/components/verilog/) the Verilog model can be checked if it can be translated successfully using the following command lines.

```
$ admsXml diode.va -e qucsVersion.xml -e qucsMODULEcore.xml
[info] admsXml-2.2.4 Oct 18 2006 19:50:46
[info] diode.core.cpp and diode.core.h: files created
[info] elapsed time: 0.0339946
[info] admst iterations: 4146 (4146 freed)
$ admsXml diode.va -e analogfunction.xml
[info] admsXml-2.2.4 Oct 18 2006 19:50:46
[info] diode.analogfunction.h created
[info] diode.analogfunction.cpp created
[info] elapsed time: 0.0262961
[info] admst iterations: 3919 (3919 freed)
```

These command lines create the files for the model evaluation code. The file names (diode.*) are due to the name of the module contained in the Verilog-AMS source file.

Additionally the source code must be changed in some more locations.

- src/components/component.h
  In this file it is necessary to add the line

```
#include "verilog/diode.core.h"
```

- src/components/component_id.h
  The file contains unique component identifiers. It is necessary to add the Verilog modules name.

```
enum circuit_type {
  CIR_UNKNOWN = -1,
  ...
  // verilog devices
  CIR_diode,
  ...
};
```

- src/input.cpp
  In order to be able to instantiate the new model the file must be modified as follows.

```
// The function creates components specified by the type of component.
circuit * input::createCircuit (char * type) {
  if (!strcmp (type, "Pac"))
    return new pac ();
  ...
  else if (!strcmp (type, "diode"))
    return new diode ();

  logprint (LOG_ERROR, "no such circuit type `%s'\n", type);
  return NULL;
}
```

- src/qucsdefs.h
  Finally the properties including their range definitions must be added to this file. The appropiate file can be obtained using the following command line.

```
$ admsXml diode.va -e qucsVersion.xml -e qucsMODULEdefs.xml
[info] admsXml-2.2.4 Oct 18 2006 19:50:46
[info] diode.defs.h: file created
[info] elapsed time: 0.0335337
[info] admst iterations: 4294 (4294 freed)
```

  The emitted file **diode.defs.h** looks like

```
/* diode verilog device */
{ "diode", 2, PROP_COMPONENT, PROP_NO_SUBSTRATE, PROP_NONLINEAR,
  {
    { "Is", PROP_REAL, { 1E-15, PROP_NO_STR }, { '[', 0, 1, ']' } },
    { "Cp", PROP_REAL, { 1E-12, PROP_NO_STR }, { '[', 0, 1, ']' } },
    { "Rs", PROP_REAL, { 1.0, PROP_NO_STR }, { ']', 0, 0, '.' } },
    { "Temp", PROP_REAL, { 300, PROP_NO_STR }, { ']', 0, 0, '.' } },
    PROP_NO_PROP },
  { PROP_NO_PROP }
},
```

representing the parameters defined in the original Verilog-AMS file. This excerpt must be included in the **src/qucsdefs.h** file into this structure:

```
// List of available components.
struct define_t qucs_definition_available[] =
{
  ...
  /* end of list */
  { NULL, 0, 0, 0, 0, { PROP_NO_PROP }, { PROP_NO_PROP } }
};
```

Finally the **Makefile.am** in the **src/components/verilog/** directory must be adjusted to make the build-system aware of the new component. There must be added

```
libverilog_a_SOURCES = ... \
        diode.analogfuncction.cpp diode.core.cpp

noinst_HEADERS = ... \
        diode.analogfuncction.h diode.core.h

VERILOG_FILES = ... diode.va

if MAINTAINER_MODE
...
diode.analogfunction.cpp: analogfunction.xml
diode.analogfunction.cpp: diode.va
        $(ADMSXML) $< -e analogfunction.xml
diode.core.cpp: qucsVersion.xml qucsMODULEcore.xml
diode.core.cpp: diode.va
        $(ADMSXML) $< -e qucsVersion.xml -e qucsMODULEcore.xml
diode.defs.h: qucsVersion.xml qucsMODULEdefs.xml
diode.defs.h: diode.va
```

```
        $(ADMSXML) $< -e qucsVersion.xml -e qucsMODULEdefs.xml
diode.gui.cpp: qucsVersion.xml qucsMODULEgui.xml
diode.gui.cpp: diode.va
        $(ADMSXML) $< -e qucsVersion.xml -e qucsMODULEgui.xml
...
else
```

in order to create the correct build rules.

If everything worked fine then the new Verilog-AMS model is now completely integrated into the analogue simulator.

Integrating the model into the GUI

Very likely as the qucs-core tree in the previous section also the qucs source tree must be configured using the –enable-maintainer-mode option.

```
$ ./configure --enable-maintainer-mode --prefix=/tmp
```

Still in the **qucs-core/src/components/verilog** directory it is now necessary to create the C++ code for the GUI using the following command line.

```
$ admsXml diode.va -e qucsVersion.xml -e qucsMODULEgui.xml
[info] admsXml-2.2.4 Oct 18 2006 19:50:46
[info] diode.gui.cpp and diode.gui.h: files created
[info] elapsed time: 0.0345217
[info] admst iterations: 4146 (4146 freed)
```

Both the created files **diode.gui.cpp** and **diode.gui.h** should be copied to the **qucs/qucs/components** directory.

Depending on the type of device several changes must be applied to these files. The constructor will basically contain the properties of the device as they are going to occur in the component property dialog as depicted in fig. 2. Check these if they are complete or if some can be left.

```
diode::diode()
{
  Description = QObject::tr ("diode verilog device");

  Props.append (new Property ("Is", "1E-15", false,
    QObject::tr ("saturation current")));
  Props.append (new Property ("Cp", "1E-12", false,
    QObject::tr ("parallel capacitance")));
```

```
  Props.append (new Property ("Rs", "1.0", false,
    QObject::tr ("series resistance")));
  Props.append (new Property ("Temp", "300", false,
    QObject::tr ("temperature")));
  ...
}
```

The **diode::info** function should be adapted to meet the devices requirements. The **BitmapFile** should be changed to represent a correct PNG file in the **qucs/bitmaps** directory. Both the **Name** and the bitmap are going to appear in the left-hand component tab as shown in fig. 3.

```
Element * diode::info(QString& Name, char * &BitmapFile,
                      bool getNewOne)
{
  Name = QObject::tr("Diode");
  BitmapFile = "diode";

  if(getNewOne) return new diode();
  return 0;
}
```

The **diode::info_pnp()** method can be completely deleted. It can be necessary for bipolar transistors where two types of devices could be displayed (npn and pnp type). The method must also be deleted from the header file. Also, in this case, the new diode class inherits **Component** instead of **MultiViewComponent**.

The **diode::createSymbol()** method must be adapted to represent a valid schematic symbol. Anyway, the default code can be used as a template.

```
void diode::createSymbol()
{
  // normal bipolar
  Lines.append(new Line(-10,-15,-10, 15,QPen(QPen::darkBlue,3)));
  Lines.append(new Line(-30,  0,-10,  0,QPen(QPen::darkBlue,2)));
  Lines.append(new Line(-10, -5,  0,-15,QPen(QPen::darkBlue,2)));
  Lines.append(new Line(  0,-15,  0,-30,QPen(QPen::darkBlue,2)));
  Lines.append(new Line(-10,  5,  0, 15,QPen(QPen::darkBlue,2)));
  Lines.append(new Line(  0, 15,  0, 30,QPen(QPen::darkBlue,2)));

  // substrate node
  Lines.append(new Line(  9,  0, 30,  0,QPen(QPen::darkBlue,2)));
  Lines.append(new Line(  9, -7,  9,  7,QPen(QPen::darkBlue,3)));

  // thermal node
```

```
    Lines.append(new Line(-30, 20,-20, 20,QPen(QPen::darkBlue,2)));
    Lines.append(new Line(-20, 17,-20, 23,QPen(QPen::darkBlue,2)));

    // arrow
    if(Props.getFirst()->Value == "npn") {
      Lines.append(new Line( -6, 15,  0, 15,QPen(QPen::darkBlue,2)));
      Lines.append(new Line(  0,  9,  0, 15,QPen(QPen::darkBlue,2)));
    } else {
      Lines.append(new Line( -5, 10, -5, 16,QPen(QPen::darkBlue,2)));
      Lines.append(new Line( -5, 10,  1, 10,QPen(QPen::darkBlue,2)));
    }

    // terminal definitions
    Ports.append(new Port(  0,-30)); // collector
    Ports.append(new Port(-30,  0)); // base
    Ports.append(new Port(  0, 30)); // emitter
    Ports.append(new Port( 30,  0)); // substrate
    Ports.append(new Port(-30, 20)); // thermal node

    // relative boundings
    x1 = -30; y1 = -30;
    x2 =  30; y2 =  30;
  }
```

In order to test the component in the GUI it is necessary to modify some more source code files.

- `qucs/qucs.cpp`
  In this file the new Verilog device will be made available to the component tab. If a new non-linear device is added then place it here:

  ```
  pInfoFunc nonlinearComps[] =
    { ... &diode::info, 0};
  ```

- `qucs/components/component.cpp`
  The new verilog device must be loadable in the GUI. So in the function **getComponentFromName()** add this:

  ```
  Component* getComponentFromName(QString& Line)
  {
    ...
    case 'd' : if(cstr == "iode") c = new diode();
          break;
    ...
  ```

```
  return c;
}
```

- `qucs/components/components.h`
  In the component header file it is necessary to add:

  ```
  #include "diode.h"
  ```

In order to make the build-sytem aware of the new Verilog model the file **qucs/-components/Makefile.am** must be modified in the following way.

```
libcomponents_a_SOURCES = ... \
  diode.gui.cpp

noinst_HEADERS = ... \
  diode.gui.h
```

With these steps the component is now fully integrated into the GUI.

Implemented devices

The following sections presents the models already added in Qucs including some test schematics and simulation results.

HICUM/L2 v2.11 model

The name HICUM was derived from HIgh-CUrrent Model, indicating that HICUM initially was developed with special emphasis on modelling the operating region at high current densities which is very important for certain high-speed applications.

The Verilog-AMS source code can be obtained from `http://www.iee.et.tu-dresden.de/iee/eb/hic_new/hic_source.html`. The model used in Qucs was a bit modified due to some ADMS limitations.

Some schematics have been setup to verify that the model emits basically the correct output values and the source code translations worked properly.

DC simulation

The setup for the forward Gummel plot is depicted in fig. 4. The base-emitter voltage is swept ($V_{BE} = 0.2 \ldots 1.2$) with a constant voltage across the second diode $V_{BC} = 0$.

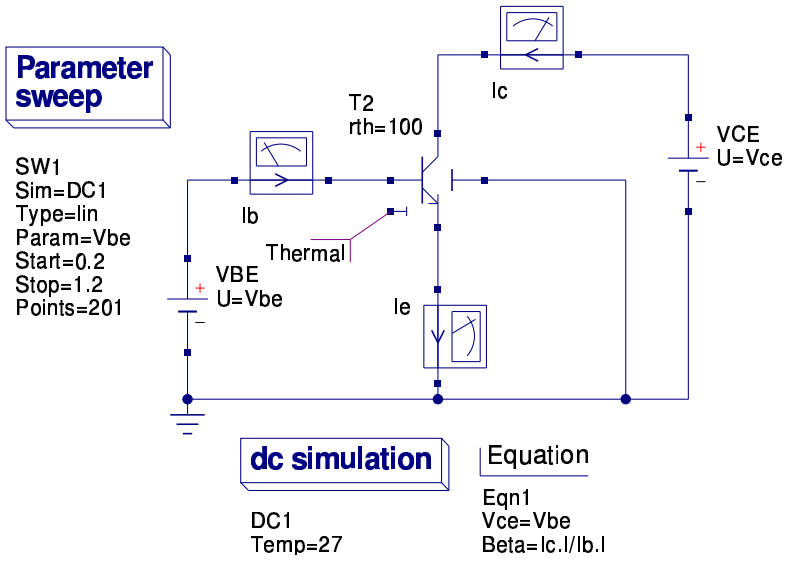

Figure 4: forward Gummel plot schematic for HICUM/L2 v2.11 model

In fig. 5 the logarithmic Gummel plot is shown including the ratio between the base current and collector current on the secondary axis.

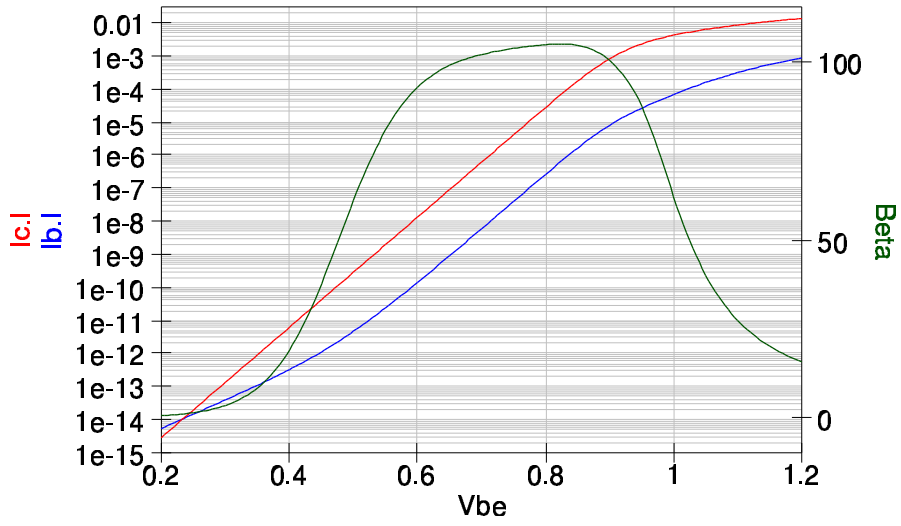

Figure 5: forward Gummel plot for HICUM/L2 v2.11 model

DC simulation

In fig. 6 the schematic for the output characteristics of the HICUM model is shown. The 2-dimensional sweep describes the function

$$I_C = f\left(V_{CE}, V_{BE}\right) \tag{4}$$

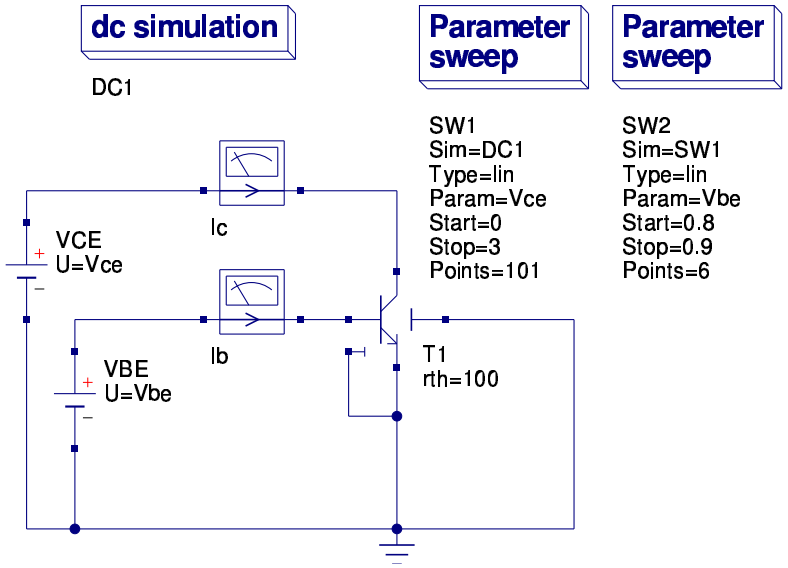

Figure 6: output characteristics schematic for HICUM/L2 v2.11 model

Figure 7 shows the results of the DC simulations for the output characteristics of the bipolar transistor.

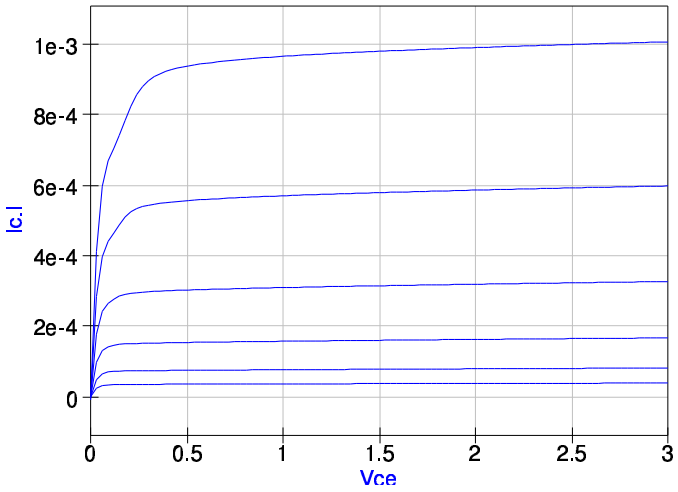

Figure 7: output characteristics plot for HICUM/L2 v2.11 model

AC simulation

Figures 8 and 9 depict the schematic and diagrams for an AC simulation in a given bias point. The current gains magnitude and phase are shown as well as the characteristics of the small signal base and collector current in the complex plane (polar plot).

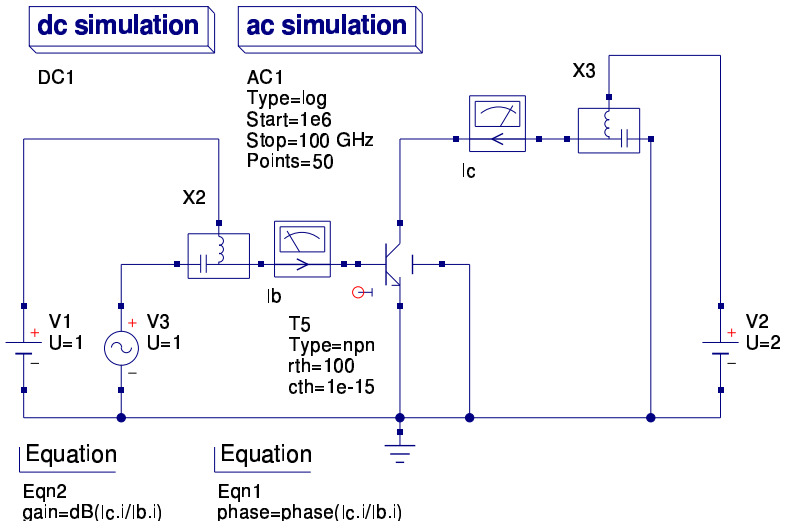

Figure 8: AC simulation schematic for HICUM/L2 v2.11 model

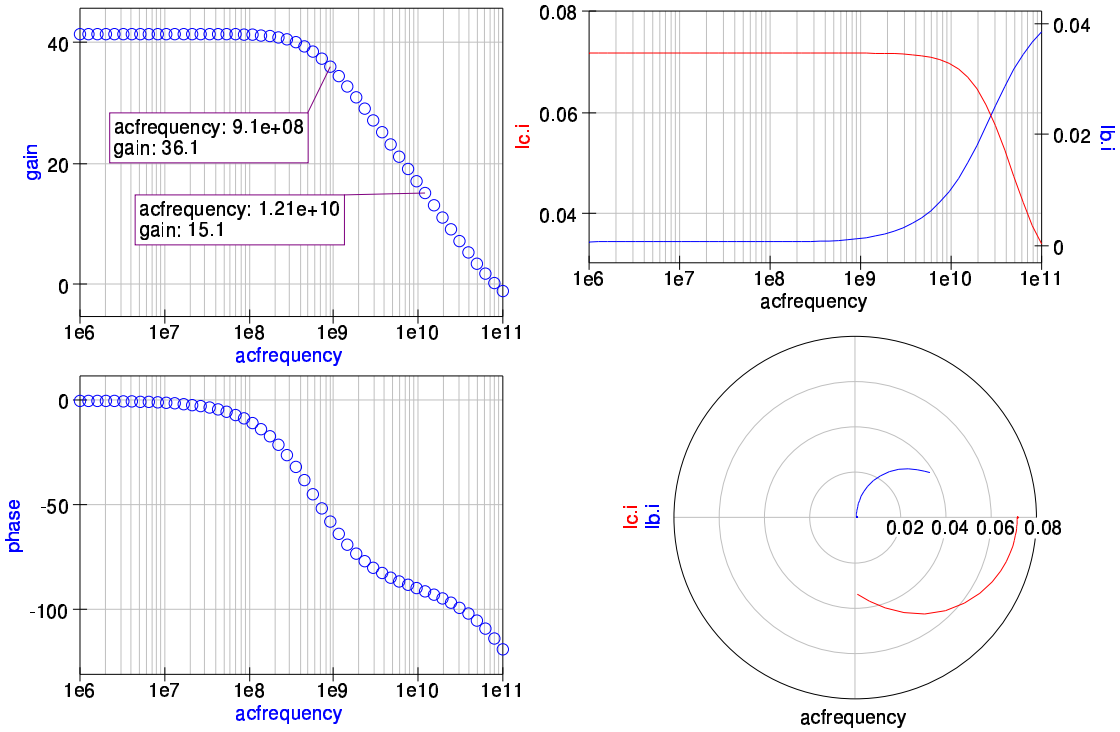

Figure 9: AC simulation plot for HICUM/L2 v2.11 model

S-parameter simulation

In the figures 10 and 11 a two-port S-parameter simulation schematic (including noise) as well as the results are shown. The four S-parameters are displayed in two Polar-Smith combi diagrams. The noise figure as well as the minimal noise figure are displayed in a logarithmic Cartesian diagram.

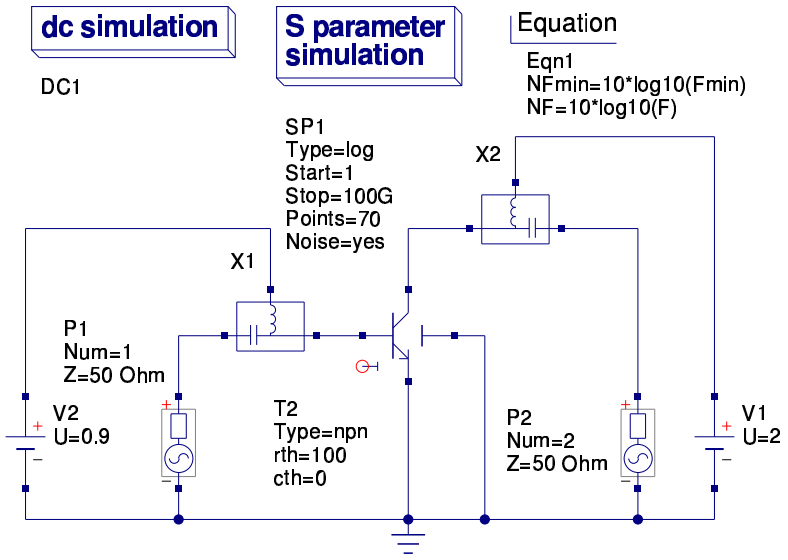

Figure 10: S-parameter simulation schematic for HICUM/L2 v2.11 model

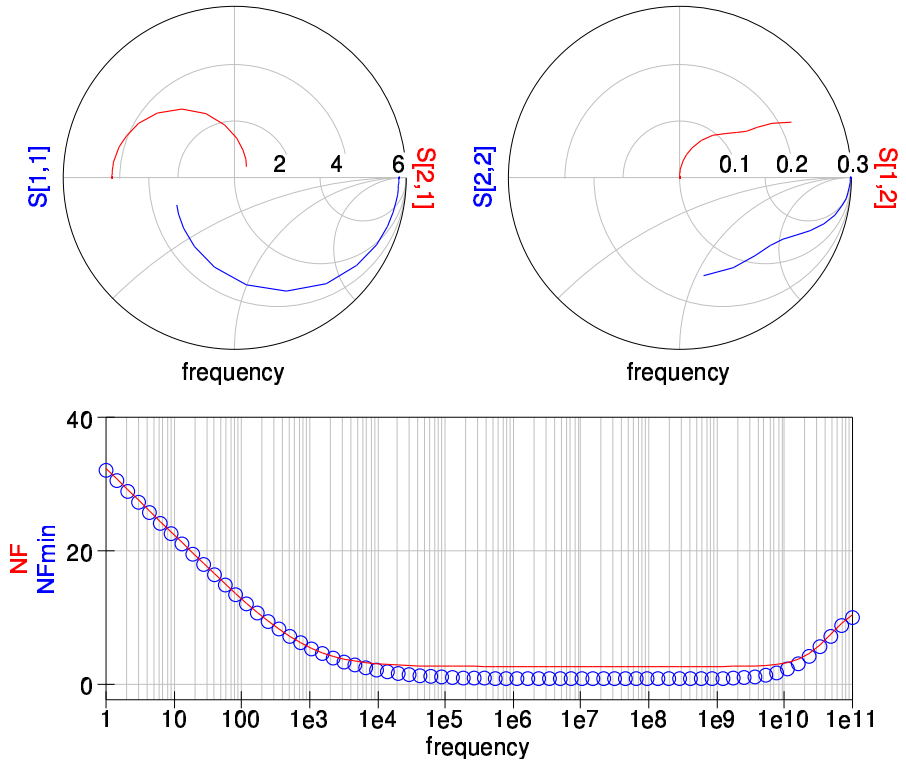

Figure 11: S-parameter simulation plot for HICUM/L2 v2.11 model

Transient simulation

In the schematic in fig. 12 a current pulse is fed into the base of the transistor. Varying the input capacitance changes the response in the collector current as shown in fig. 13.

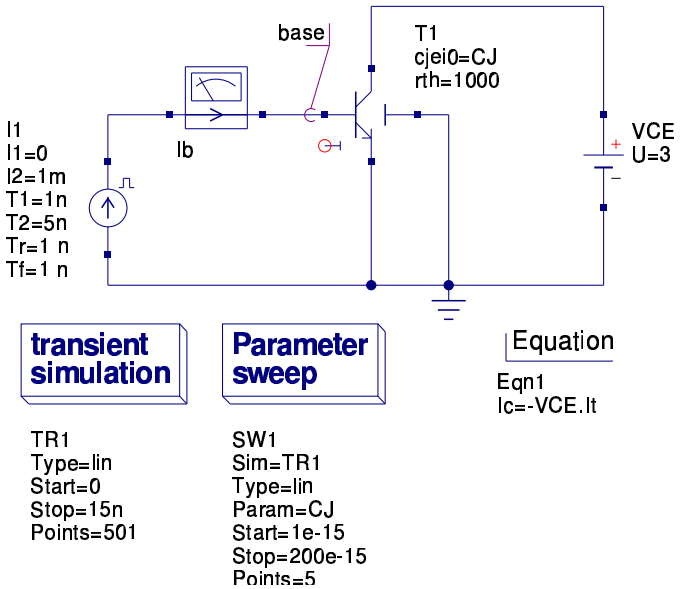

Figure 12: Transient simulation schematic for HICUM/L2 v2.11 model

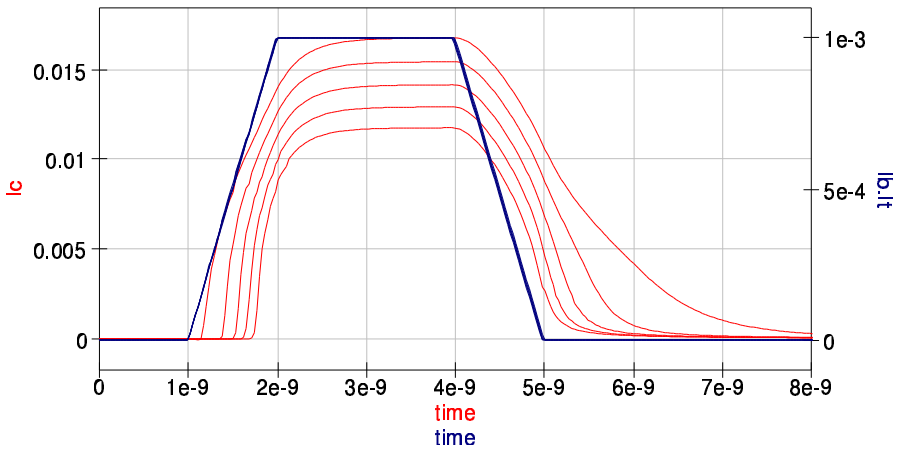

Figure 13: Transient simulation plot for HICUM/L2 v2.11 model

FBH-HBT model version 2.1

The HBT (Hetero Bipolar Transistor) model developed by Matthias Rudolph at the FBH (Ferdinand-Braun-Institut für Hochfrequenztechnik) is available in the Internet at `http://www.designers-guide.org/VerilogAMS`.

DC simulation

Forward Gummel plot.

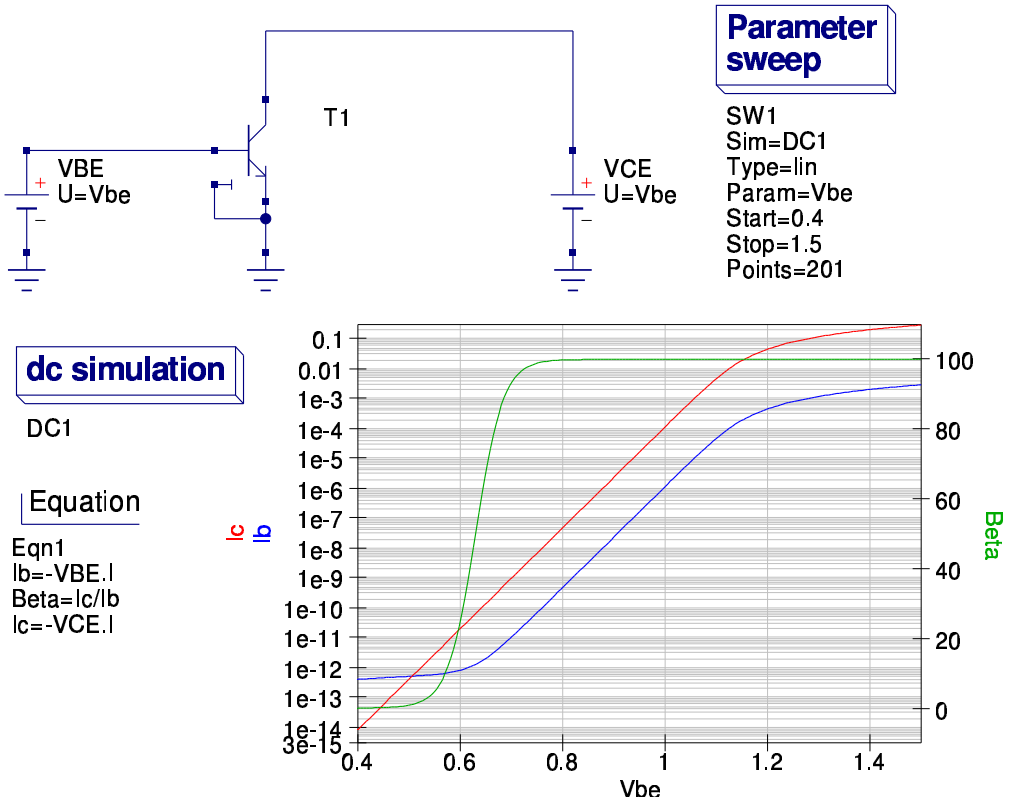

Figure 14: forward Gummel plot schematic for HBT model

DC simulation

Output characteristics.

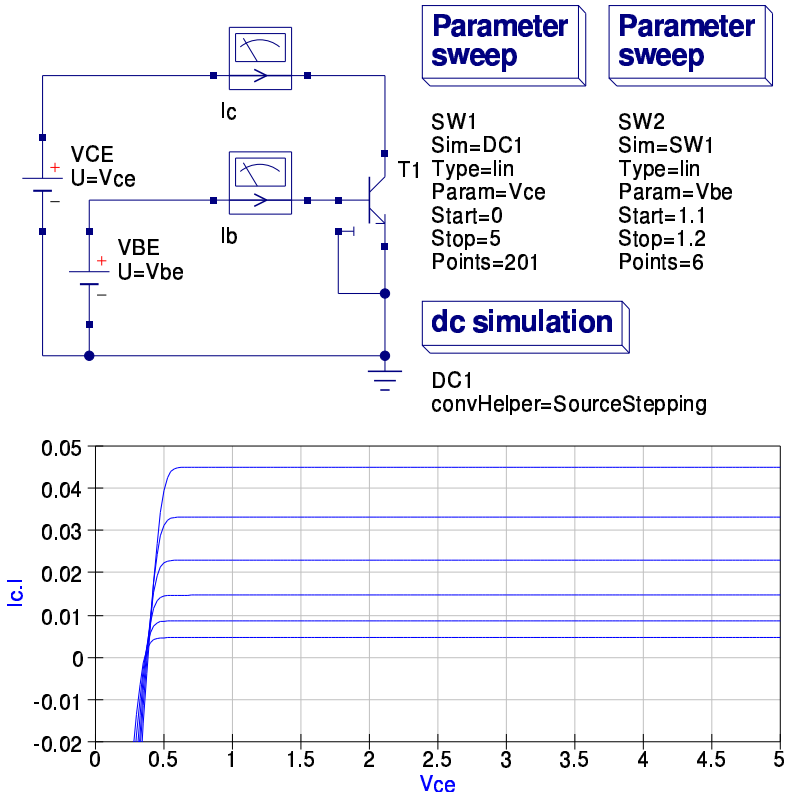

Figure 15: output characteristics schematic for HBT model

End Note

The preferred way to add a new Verilog model to Qucs is certainly to prepare a model, check if it is accepted by ADMS and finally ask the maintainers to integrate it. Until there is no possibility to load device modules dynamically (which is on the TODO list) too many hand-made changes have to be done to automate this process. The dynamic modules require changes in the simulator API as well in the GUI code.

Furthermore work is going to be continued on ADMS itself as well as on the **admst** scripts.

The authors would like to thank Hélène Parruitte for the initial implementation of the Verilog-AMS interface in Qucs. Also thanks go to the company Xmod Technologies (see `http://www.xmodtech.com`) allowing her to work on such an interface and finally to share the outcome of her internship under the GPL.