6

Principles of Active Electrical Circuit Modeling

Preview

In this chapter, we discuss the tools and techniques of today's professional circuit designers. We discuss how SPICE works, and what additional tools exist that support SPICE, such as the PROBE feature to view simulation trajectories graphically, or Workview [6.11] for schematic capture. We shall then explore the pro's and con's of using DYMOLA [6.3] for electronic circuit modeling.

6.1 Topological Modeling

As we have shown in Chapter 3, it is rather inconvenient to manually derive a state–space description for even fairly simple passive electrical circuits. We have seen that it may not be desirable at all to even attempt to generate such a state–space description due to the frequent algebraic loops and structural singularities inherent in most practical electrical circuits.

For this reason, circuit designers prefer to model their circuits by specifying a *topological description.* Let us revisit the simple passive circuit that was discussed in Chapter 3. It is shown once more in Fig.6.1.

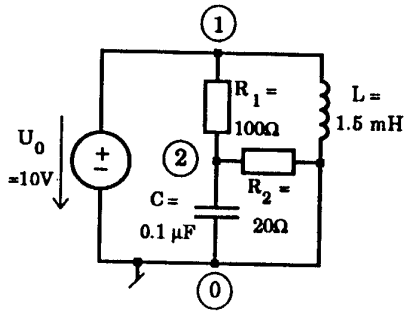

Figure 6.1. Example of a passive circuit

Using the topological modeling approach, we simply number all nodes of the circuit, and describe the circuit topology by specifying terminal nodes for each circuit element. This is the approach taken by all current circuit analysis programs. The most common among those is SPICE. Various SPICE dialects exist which have partly been coded in Fortran and partly in C. Among those, we currently recommend PSpice [6.9] which runs both on main frames and on PC's, and which works fairly well. The above shown circuit can be modeled in PSpice in the following manner:

```
Simple Passive Electrical Circuit
  R1  1  2  100
  R2  2  0  20
  L   1  0  1.5M
  C   2  0  0.1U
  V0  1  0  10
.OP
.END
```

The ".OP" statement describes the simulation experiment, i.e., the type of analysis that is to be performed on the circuit (in this case, a computation of the DC steady–state value). The first character of the element names is semantically significant. It determines the type of circuit element being used. Node numbers are positive integers except for the ground node which is always declared to be node 0. A character immediately following a numeric constant (an element value) denotes a scaling factor. For example, "M" stands for "milli", and "U" stands for "micro". SPICE uses the *branch admittance matrix* approach to convert the topological circuit description into an implicit matrix description in the way described in Chapter 3.

6.2 Models of Active Devices in SPICE

Circuits that are of any practical interest today are always heavily non–linear. This is due to the fact that most circuits today are built as *integrated circuits*, and the electrical phenomena that occur in a p–n junction are governed by heavily non–linear equations.

Let us look at a simple p–n junction first. Such a junction is shown in Fig.6.2.

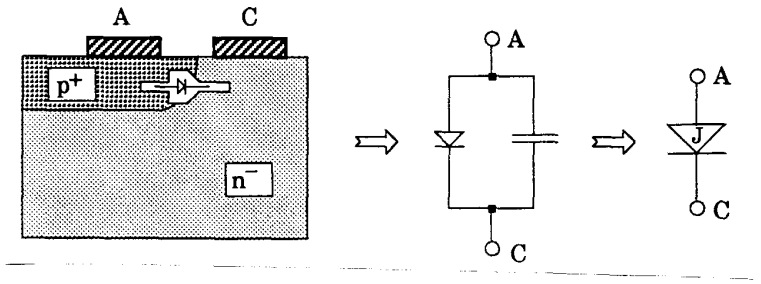

Figure 6.2. Model of a p–n junction

Every p–n junction can be described as a *diode* which has its anode on the p–side and its cathode on the n–side of the junction, i.e., the junction is forward biased if the potential on the p–side of the junction is higher than that on the n–side. In this case, current flows through the junction. The junction is reverse biased otherwise. The relation between current through and voltage across a diode can be described by the following equation:

$$i_d = I_s[\exp(\frac{u_d}{V_T N_d}) - 1] + G_{min} u_d \tag{6.1}$$

I_s is the saturation current of the diode, V_T is the thermal voltage, $V_T = \frac{kT}{q}$, and N_d is the emission coefficient. The second term denotes a resistive leakage current.

Especially for reverse biased junctions, electrical charge will build up along the junction since no current can flow. This charge can be described as a non–linear capacitance that is connected in parallel to the diode. In Chapter 3, we saw that the relation between current through and voltage across a capacitor is:

$$i_c = C\frac{du_c}{dt} \tag{6.2}$$

However, this equation is only correct for linear capacitors. If the capacitance value changes over time, this equation must be replaced by the more general equation:

$$i_c = \frac{d}{dt}(C \cdot u_c) = \frac{dq_c}{dt} \tag{6.3}$$

where $q_c = C \cdot u_c$ is the electrical charge stored in the capacitance. Even more generally, the relation between current and voltage in the junction capacitance can be described as:

$$i_c = \frac{dq_c}{dt}, \quad q_c = f(u_c) \tag{6.4}$$

The charge in the junction capacitance can be approximately described by the following equation:

$$q_c = \tau_d i_d + \frac{\phi_d C_d[1 - (1 - \frac{u_d}{\phi_d})^{1-m_d}]}{1 - m_d} \tag{6.5}$$

where i_d stands for the current through the junction diode, and is in itself a function of u_d as stated in eq(6.1). τ_d is the time constant of the capacitance, and the parameters ϕ_d, C_d, and m_d model second order transient effects.

Let us now look at transistors. A bipolar junction transistor (BJT) basically consists of two p–n junctions: a base–collector junction, and a base–emitter junction. Fig.6.3 shows two ways that BJT's can be built.

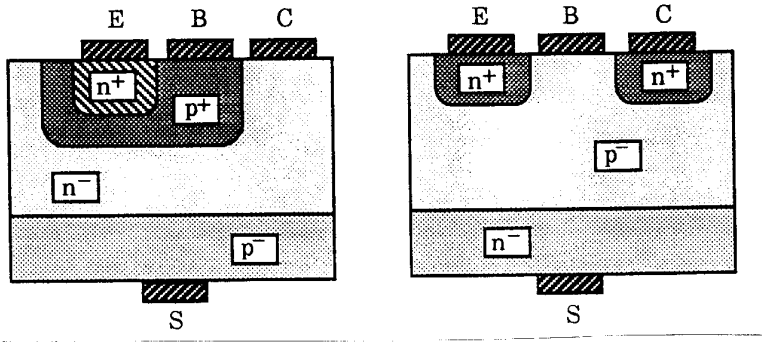

Figure 6.3. Vertical and lateral NPN transistors

The two p–n junctions can either be embedded into each other as on the left of Fig.6.3. Such transistors are commonly referred to as *vertically diffused transistors.* Alternatively, they can be placed next to each other. Such transistors are called *laterally diffused transistors.* Both transistor types shown in Fig.6.3 are NPN transistors (identifying the doping of the three major regions (N:Emitter, P:Base, and N:Collector). PNP transistors look exactly the same except that all doping concentrations are reversed. The NPN transistor can be modeled as shown in Fig.6.4.

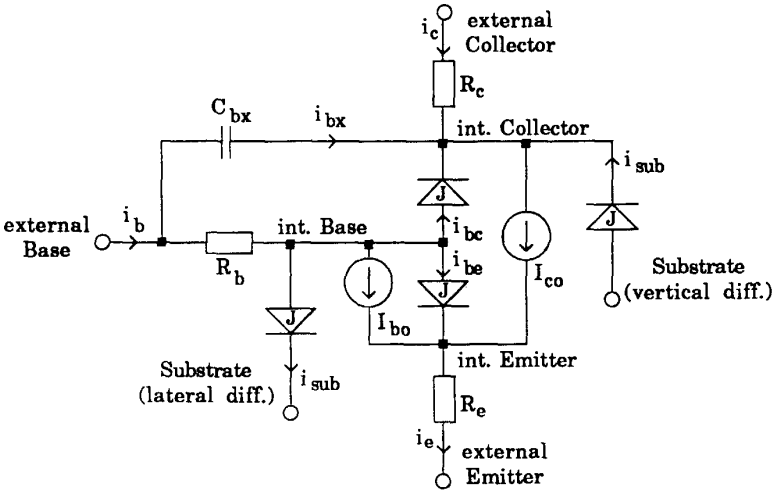

Figure 6.4. Model of vertical and lateral NPN transistors

In the vertically diffused BJT, the substrate is connected to the collector, while in the laterally diffused BJT, it is connected to the base. The C_{bx} capacitance is actually a part of the junction capacitance of the base–collector junction diode. By splitting this capacitance between the external base and the internal base, the physically distributed junction charge can be represented a little more realistically. The model for PNP transistors looks exactly the same except that all diode polarities are reversed. The two dependent current sources are modeled in the following way:

$$I_{Co} = \frac{i_{be} - i_{bc}}{q_b} - \frac{i_{bc}}{B_R} - i_{bcn} \tag{6.6a}$$

$$I_{Bo} = \frac{i_{be}}{B_F} + i_{ben} + \frac{i_{bc}}{B_R} + i_{bcn} - \frac{i_{be}}{q_b} \tag{6.6b}$$

where i_{be} and i_{bc} denote the diode currents through the base–emitter and base–collector junction diodes, i_{ben} and i_{bcn} denote the same quantities once more, but this time using altered saturation currents and modified emission coefficients, and B_F and B_R are the ideal forward and backward β coefficients which denote the DC current gain factors from the base to the emitter and to the collector, i.e., $\frac{I_E}{I_B}$ and $\frac{I_C}{I_B}$. q_b is the base charge, and is in itself a computed quantity which, in SPICE, is approximated by the following equation:

$$q_b = \frac{q_1}{2}(1 + \sqrt{1 + 4q_2}) \tag{6.7}$$

where

$$q_1 = \frac{1}{1 - \frac{v_{bc}}{V_{AF}} - \frac{v_{be}}{V_{AR}}} \tag{6.8a}$$

$$q_2 = \frac{i_{bc}}{I_{KR}} + \frac{i_{be}}{I_{KF}} \tag{6.8b}$$

The base resistance is the most important resistance in the BJT model. Consequently, it is modeled more accurately than the other two resistances. The base resistance depends on the base current. The following equation is used to model the current dependence of the base resistance in SPICE:

$$r_{bb} = R_{BM} + 3(R_B - R_{BM})\frac{\tan(z) - z}{z(\tan(z))^2} \tag{6.9}$$

where:

$$z = \frac{-1 + \sqrt{\frac{1 + 144 i_b}{\pi^2 I_{RB}}}}{\frac{24}{\pi^2}\sqrt{\frac{i_b}{I_{RB}}}} \tag{6.10}$$

Not all SPICE dialects use exactly the same equations. The equations presented in this text are those of the BBSPICE [6.1] dialect, an HSPICE [6.8] offspring, since this is the only version of SPICE for which I have source code available. Unfortunately, few SPICE

manuals are explicit in these matters, and many are inaccurate. It is therefore dangerous to rely in these matters on information provided in a user's manual rather than to extract it from the source code directly.

The equations presented in this chapter are in fact only a subset of those that are actually used in the SPICE model. For example, eq(6.10) won't work if the parameter I_{RB} takes on its default value of 0.0. In that case, SPICE automatically switches over to another simpler equation to approximate the base resistance, namely:

$$r_{bb} = R_{BM} + \frac{R_B - R_{BM}}{q_b} \qquad (6.9^{alt})$$

where q_b is the previously introduced base charge.

In fact, while the BJT model used by BBSPICE is the Gummel–Poon model, BBSPICE will automatically revert to the simpler Ebers–Moll model if default values are used for the secondary device parameters.

Also, it can easily happen that the denominator of eq(6.8a) becomes zero in which case this equation would blow up. BBSPICE automatically flattens out denominators in the vicinity of zero, and limits expressions to be exponentiated in size to prevent the model from blowing up.

Finally, many of the model parameters are temperature dependent. Military rated devices must operate correctly between $-55°C$ and $+120°C$. It is thus important that the circuit simulator can take effects of temperature variation into account. In BBSPICE, temperature variation is modeled by a quadratic approximation. Many of the device parameters have temperature coefficients associated with them. They all work basically in the same way. For instance, the temperature variation of the base resistance is modeled by:

$$\Delta Temp = Temp - T_{Room} \qquad (6.11a)$$

$$r_b = R_B + T_{RB1}\Delta Temp + T_{RB2}\Delta Temp^2 \qquad (6.11b)$$

$$r_{bm} = R_{BM} + T_{RM1}\Delta Temp + T_{RM2}\Delta Temp^2 \qquad (6.11c)$$

In reality, the temperature dependent coefficients r_b and r_{bm} are plugged into eq(6.9) and eq(6.9^{alt}), and not the user supplied constant coefficients R_B and R_{BM}.

BBSPICE offers an analysis statement of the type:

```
.TEMP  -55  -30  -5  20  45  70  95  120
```

which allows us to repeat whatever analysis was requested for a number of different temperature values. All temperature dependent parameters are updated between simulation runs.

It would not be very practical if all model parameters would have to be user specified for every single transistor in the circuit separately. For example, in BBSPICE, BJT's contain 53 different model parameters that can be user specified. Since typical circuits are fabricated as integrated circuits on one chip, and since for economic reasons not too many different processes can be involved in the fabrication of a single chip, chances are that most of the BJT's in a circuit are very similar, maybe except for the *area* that the emitter occupies (which, in turn, will influence many of the other parameters, i.e., the resistances will be divided by the area while the capacitances will be multiplied by the area; however, most SPICE dialects provide for one *Area* parameter). To avoid this problem, SPICE allows us to group sets of device parameters into a *.MODEL* statement such as:

```
.MODEL  PROC35.N  NPN
+ IS=1.1FA  BF=190  BR=.1  EG=1.205612  ISS=0
+ ISE=0  ISC=0  NE=1.5  NC=2  BULK=SUB
+ VAF=110  VAR=0  IKF=3.6MA  IKR=0  SUBS=1
+ RB=1109.9  IRB=5.63MA  RBM=368.4  RE=13.3  RC=750
+ XTB=.006  XTI=2.33  TRE1=.0005  TRB1=.005  TRC1=.005
+ TRM1=.005  TRE2=0  TRB2=0  TRC2=0  TRM2=0
+ CJE=.597PF  VJE=.77  MJE=.3
+ CJC=.36PF  VJC=.64  MJC=.425
+ CJS=0  VJS=.75  MJS=0  XCJC=1  TLEV=1
+ NF=1  NR=1  NS=1  LEVEL=2  VTF=0  AF=1
+ TF=50PS  TR=1US  XTF=1  ITF=36MA  PTF=0  KF=.16F


.MODEL  PROC35.P  PNP
+ IS=1.1FA  BF=125  BR=.1  EG=1.205612  ISS=0
+ ISE=0  ISC=0  NE=1.5  NC=2  BULK=SUB
+ VAF=30  VAR=0  IKF=3.6MA  IKR=0  SUBS=-1
+ RB=778  IRB=1.93MA  RBM=57.6  RE=23.3  RC=450
+ XTB=.006  XTI=2.33  TRE1=.0005  TRB1=.005  TRC1=.005
+ TRM1=.005  TRE2=0  TRB2=0  TRC2=0  TRM2=0
+ CJE=.652PF  VJE=.77  MJE=.3
+ CJC=.055PF  VJC=.64  MJC=.425
+ CJS=0  VJS=.75  MJS=0  XCJC=1  TLEV=1
+ NF=1  NR=1  NS=1  LEVEL=2  VTF=0  AF=1
+ TF=100PS  TR=1US  XTF=1  ITF=90MA  PTF=0  KF=.16F
```

In SPICE, lines starting with a "+" are continuation lines. Models can then be invoked by referring to their model identifier. For example, models of the two types *PROC35.N* and *PROC35.P* can be invoked through the statements:

```
Q120  7  21  15  PROC35.N
Q121  9  42  31  PROC35.P
```

where the character "Q" in the element name indicates the BJT. A small subset of the model parameters can also be specified during the element call itself in which case this value supersedes the value specified in the *.MODEL* statement, which in turn supersedes the default value. Typically, the *area* parameter will be specified during the element call since it varies from one transistor to the next while other process parameters remain the same.

How can we identify a decent set of model parameters for a given transistor? Two answers to this question can be given. On the one hand, some companies sell model libraries for various SPICE dialects that contain sets of model parameters for most of the commercially available semiconductor devices. One such product is ACCULIB [6.7], a model library for the SPICE dialect ACCUSIM [6.6]. On the other hand, it is possible to buy computerized data acquisition systems which automatically produce test signals for a variety of semiconductor devices, and quickly identify a set of model parameters for the given device. One such product is TECAP [6.4]. The TECAP system consists of an HP 9000 computer, an HP 4145 semiconductor analyzer, a network analyzer, and a capacitance meter. The chip to be modeled is inserted in the system. The system then generates a series of *test signals* for the chip, and records the chip's responses. It then computes a rough first set of model parameters. Thereafter, it uses these model parameters as initial parameter values for a PSpice optimization study. Simulations are run, and the parameters are adjusted until the PSpice simulation is in good agreement with the measured characteristics. When operated by an experienced user, TECAP requires roughly 20 *min* to identify the DC parameters of a BJT transistor, and another 30 to 40 *min* to identify its AC parameters.

In this section, we have used the BJT to explain, by means of an example, how involved the active device models in modern circuit analysis programs are. We chose the BJT because its model is quite a bit simpler than the MOSFET model. However, the modeling principles are the same.

One disadvantage of the intrinsic "model" concept employed in SPICE is the fact that most device models are very complex, and they are not transparent to the user. Furthermore, most SPICE versions (exceptions: BBSPICE [6.1] and HSPICE [6.8]), provide the engineer with insufficient access to the internal voltages, currents, and circuit element parameters of the models. Consequently, it is often quite difficult to interpret effects shown by the simulation since the equations used inside the models are not truly understood by and often not known to the design engineers. Also, the SPICE models are rather difficult to maintain and to upgrade. Recently, I added a four terminal GaAs MESFET model to BBSPICE that we had received from another company. It took me a whole week and required changes to 25 subroutines to integrate this model with the BBSPICE software.

6.3 Hierarchical Modeling

In many circuit simulation studies, it is not practical to model each circuit down to the level of the available SPICE models (the BJT's, JFET's, and MOSFET's) since the same higher level components (such as a logical inverter subcircuit) is reused in the circuit several times. It would, of course, be feasible to implement such a subcircuit as a new SPICE model, and this is certainly the right approach if this same component is going to be reused over and over again, such as a Zener diode. However, this approach is painfully slow and cumbersome at best.

Therefore, SPICE also offers the possibility to declare an ensemble of circuit elements as a *subcircuit*, and to reuse it thereafter as often as needed. Subcircuits are *macros* as introduced in Chapter 5. However, at the abstraction level of a topological circuit description, macros *are* truly modular which was not so in the case of state–space descriptions.

The following code shows a typical BBSPICE subcircuit which describes a generic gain stage of an operational amplifier:

```
.SUBCKT GAIN 100 101 1 2 RG = 500
*INPUT CIRCUIT
RS   100   3   50Ohm
RIN  3     0   50Ohm
*LOAD CIRCUIT
RO   5     101 50Ohm
RL   101   0   50Ohm
*GAIN SET RESISTORS
RF1  5     26  500Ohm
RF2  26    25  500Ohm
RF3  4     25  500Ohm
CP1  25    0   1.4pF
CP2  26    0   1.4pF
RE   4     0   RG
*AMPLIFIER
XAMP 3 4 5 1 2 SERSHNT
.ENDS GAIN
```

which then can be invoked through the statement:

```
XT1 50 51 7 8 GAIN RG = 78.95
```

The node numbers of the subcircuit definition are formal positional parameters that are replaced by the actual node numbers during the invocation of the subcircuit. Other (internal) nodes of the subcircuit are local variables of the macro, and are unaccessible from the outside. *RG* is a subcircuit parameter which can be assigned a default value in the subcircuit declaration (in our case 500). The default value can be overridden during the subcircuit invocation process. In our example, *RG* receives now a value of 78.95. The "X" character in the element name (XT1) indicates that this "element" is in fact not an element, but a subcircuit call. Subcircuits can be nested. In our example, the subcircuit *GAIN* calls upon another subcircuit of type *SERSHNT*.

In practice, the subcircuit concept in SPICE is a little clumsy, and subcircuits often create problems. On the one hand, many SPICE versions give the user access to the top level of the hierarchy only, i.e., the node voltages and branch currents that are *internal* to a subcircuit cannot be displayed on output. On the other hand, most SPICE versions do not allow the user to specify an initial nodeset for internal nodes of subcircuits (cf. the subsequent discussion of *ramping*). This often creates serious problems with DC–convergence. The cause of these difficulties is the fact that, in the original Berkeley SPICE, nodes had been *numbered* rather than *named*. Some of

the newer versions of SPICE have solved this problem by introducing node names. In such versions, internal nodes of subcircuits can be accessed using a dot–notation. For example, the voltage of the internal node *emitter* of the subcircuit *opamp* can be addressed as *V(opamp.emitter)*, and the current through the base resistance R_{bb} of the subcircuit *opamp* can be addressed as *I(opamp.rbb)*.

6.4 Transient Analysis in SPICE

Transient analysis, in a circuit analysis program, corresponds to the previously introduced mechanism of *system simulation*. Starting from a given *initial condition*, the trajectory behavior of the system over time is determined. SPICE has a peculiar way of handling initial conditions. In SPICE, it is assumed that the initial value of all sources have been present for an infinite amount of time before the simulation starts. Let me explain this concept.

In a linear system of the type:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{6.12}$$

we can set $\mathbf{u}$ to its initial value $\mathbf{u_0}$, and compute a steady–state solution. In steady–state, all derivatives have died out, i.e., $\dot{\mathbf{x}} = 0.0$. Therefore, we find:

$$\mathbf{x_0} = -\mathbf{A}^{-1}\mathbf{B}\mathbf{u_0} \tag{6.13}$$

In a non–linear system of the type:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{6.14}$$

we must solve an implicit set of non–linear equations of the form

$$\mathbf{f}(\mathbf{x_0}, \mathbf{u_0}, t) = 0.0 \tag{6.15}$$

for the unknown vector $\mathbf{x_0}$. This is what SPICE attempts when it computes an "OP–point".

SPICE tackles this problem by assuming an initial value for $\mathbf{x_0}$, linearizing the circuit around this assumed value, and then computing a better estimate for $\mathbf{x_0}$ using the approach of eq(6.13). This scheme is called *Newton–Raphson iteration.*

Let us look once more at the junction diode example. The current through the diode is a non–linear function of the voltage across the diode. The algorithm starts with an initial voltage which the user can specify in a so–called "nodeset", and which is assumed to be zero by default. Then, all non–linear functions are linearized around the present working point as shown in Fig.6.5.

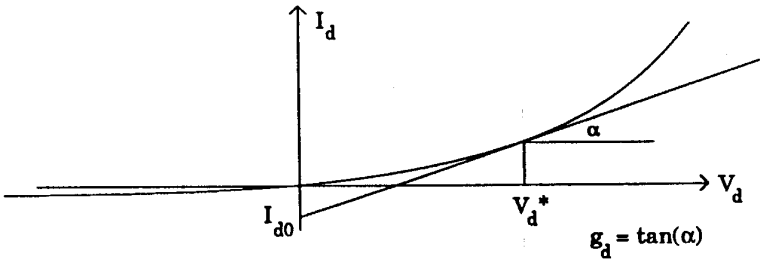

Figure 6.5. Linearization of non–linear diode characteristic

In the linearized model, the diode is replaced by a conductance $g_d = \frac{\partial i_d}{\partial u_d}$, and a current source I_{d0}. Now, we can use this model to solve the branch admittance matrix equation for a new, and hopefully improved, nodeset. The iteration continues until the difference between subsequent nodesets has become negligible.

Unfortunately, in a non–linear circuit, no guarantee can be given that this algorithm will converge to the correct OP–point, or will converge at all. DC–convergence is one of the big problems in circuit analysis. The design engineer can help the program by specifying a good nodeset, i.e., by specifying good initial guesses for some or all of the node potentials in his or her circuit. The closer the nodeset is to the true value, the more likely it is that the iteration will converge to the desired value, and the faster the convergence will be.

Sometimes, this approach does not work since the design engineer is unable to come up with a sufficiently close nodeset for her or his circuit. In these cases, design engineers have developed a technique of "ramping up all sources". This works in the following way: We first perform an experiment (i.e., a simulation) in which all (voltage and current) sources are initially set to zero, and in which all active devices (transistors, diodes) have been switched *off* (one of the device parameters). In this setup, the OP–point computation is trivial. All

node voltages must obviously be zero. Now, we perform a *transient analysis* (i.e., a simulation) in which we apply a ramp to all sources as shown in Fig.6.6.

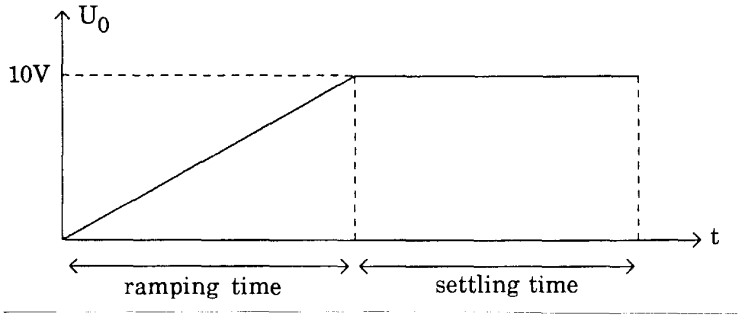

Figure 6.6. Ramping of sources in an electrical circuit

The final value of each ramp is the desired initial value for the real simulation which is then held constant for some time to allow the circuit to settle down. The final value of all node voltages can then be printed out and copied into a nodeset to be used by the subsequent simulation in which the real problem is being solved.

Some versions of SPICE (but not PSpice [6.9]) have automated this procedure. In BBSPICE [6.1], I have added a ".RAMP" statement of the form:

$$.RAMP < ramping\ time >< settling\ time >$$

Whenever the initial OP–point fails to converge, BBSPICE will automatically modify the circuit internally to the form described above, perform an invisible transient analysis, compute an initial nodeset, convert the circuit back to its initial specification, and then perform the desired analysis. This approach works very well, and the design engineers at Burr Brown are very happy with this utility.

Once the initial condition has been determined, we can proceed to compute the trajectory behavior of the system over time. Here, we also face a number of awkward problems. In past chapters, we have learned that we always like to solve differential equations for their highest derivative terms, and then integrate the derivatives into a new set of states via *numerical integration.* Unfortunately, this approach doesn't work in SPICE. Let us look once more at the junction capacitance equations, eq(6.4). We would like to solve the

differential equation for $\dot{q}_c$. However, this would make q_c a "known" variable, and we would have to solve the equation $q_c = f(u_d)$, eq(6.5), for the variable u_d. However, this cannot be done since this equation does not have an analytical inverse, and moreover, since the initial value for u_d is user specified (in the nodeset). Consequently, SPICE has no choice but to compute q_c from the non–linear equation $q_c = f(u_d)$, and then evaluate i_c using *numerical differentiation.* As will be shown in the second volume of this text, this approach is dubious from a numerical point of view unless we use an *implicit numerical differentiation scheme.*

Once all the currents have been determined, a new nodeset can be projected for Δt time units into the future.

6.5 Graphical Modeling

About three years ago, a new facet was added to industrial circuit design technology. This came with the advent of the new 386–based engineering workstations. For the first time, it was feasible to place a computer on every design engineer's desk, a computer that is not just a toy, but is capable of solving even relatively large circuit analysis problems in a decent time frame. At Burr Brown, every design engineer has now a Compaq 386 on his or her desk, and most circuit analysis problems are being solved in PSpice which runs beautifully on these machines. PSpice (and BBSPICE) offer a powerful *Probe* option which allows the design engineer to perform a transient analysis, and then look at arbitrary node voltages or branch currents interactively on her or his screen. This feature has enhanced the efficiency of circuit design drastically. Previously, the engineer had to decide beforehand which variables s/he wanted to print out, and if his or her analysis of the obtained results indicated that s/he needed more output, s/he actually was forced to rerun the entire simulation, which often required several hours of execution time on a VAX 11/750.

Model input is now also being performed graphically using schematic capture programs. At Burr Brown, we use software called Workview [6.11] which allows us to draw electrical circuits on a virtual screen. Workview is one among roughly a dozen or so similar schematic capture programs currently on the market. In Workview, the physical screen can be moved over the virtual screen (zoom and

pan). This is necessary for drawing complex circuits since it is often impossible to fit an entire circuit drawing on one physical screen. Fig.6.7 shows the Workview drawing of a simple (integrated) operational amplifier consisting of 12 BJT's, one resistor, and one capacitor.

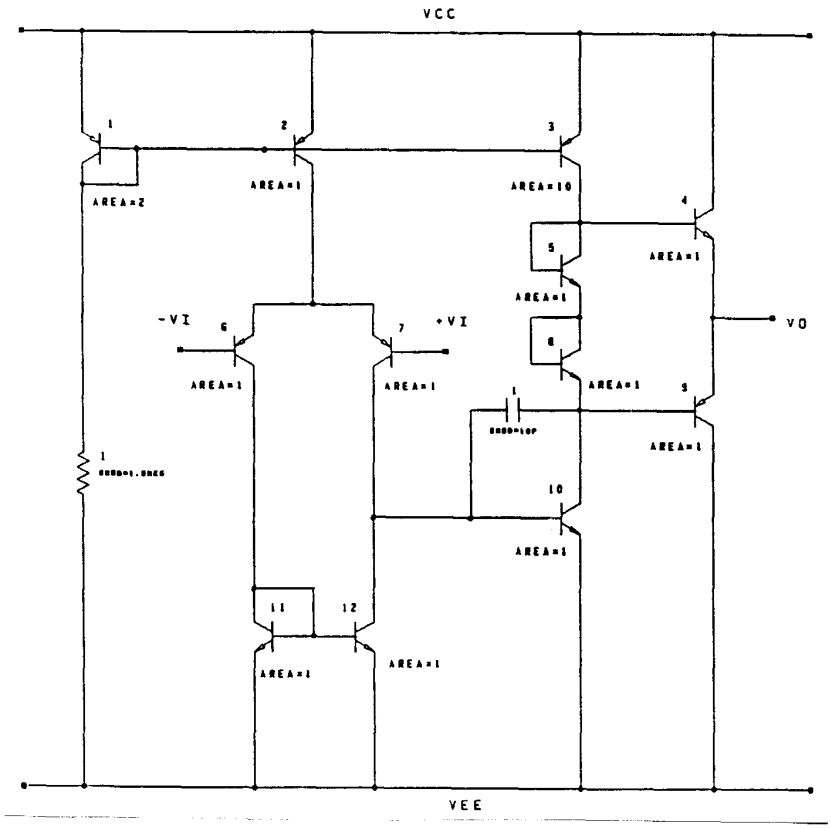

Figure 6.7. Workview drawing of an opamp

Workview enables the user to automatically generate code for PSpice. Below, the code is shown that Workview generates for the simple opamp of Fig.6.7.

```
* Project OPAMP
* WorkVIEW Wirelist Created with Version 3.0

Q12     1   2   3   3   PROC35.N   1
Q5      4   4   5   3   PROC35.N   1
Q3      4   6   7   3   PROC35.P   10
Q1      6   6   7   3   PROC35.P   2
Q2      8   6   7   3   PROC35.P   1
Q4      7   4   9   3   PROC35.N   1
Q6      2   10  8   3   PROC35.P   1
Q7      1   11  8   3   PROC35.P   1
Q8      5   5   12  3   PROC35.N   1
Q9      3   12  9   3   PROC35.P   1
Q10     12  1   3   3   PROC35.N   1
Q11     2   2   3   3   PROC35.N   1
C1      1   12  10P
R1      3   6   1.5MEG

* DICTIONARY 6
* +VI = 11
* -VI = 10
* V0 = 9
* VCC = 7
* VEE = 3
* GND = 0

.END
```

Notice that all BJT's are produced using the same process technology, i.e., the same photolithography and diffusion steps. They are all of either the *PROC35.N* type (for NPN transistors), or the *PROC35.P* type (for PNP transistors). The *area* parameter is specified on the element call statement. For example, the PNP transistor *Q3* occupies an area which is 10× as large as the area that was used to determine the values of the model parameters on the *.MODEL* statement.

Workview also supports the concept of hierarchical modeling. A subcircuit can be drawn, its terminals can be identified, and a new symbol (icon) can then be sketched which must have as many terminals as the subcircuit has, and which, from now on, represents the subcircuit, and can be used as a circuit element in the next higher level of the modeling hierarchy. In Fig.6.8, a new icon was constructed to denote the opamp, and the opamp is being used as a modeling element in a higher level circuit.

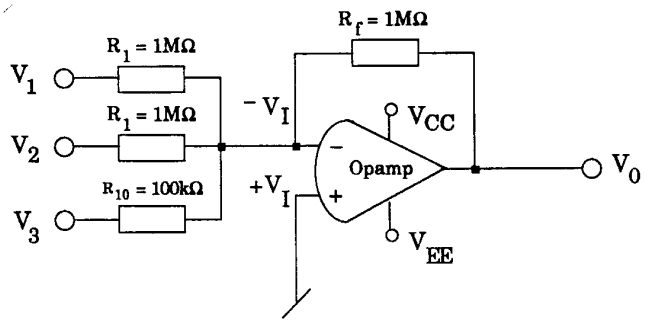

Figure 6.8. Drawing of an analog signal adder/inverter

Workview can be requested to map its modeling hierarchies into PSpice's subcircuits. Alternatively, the graphical modeling hierarchy can also be flattened out. We shall discuss the pro's and con's of hierarchy flattening in Chapter 15 of this text.

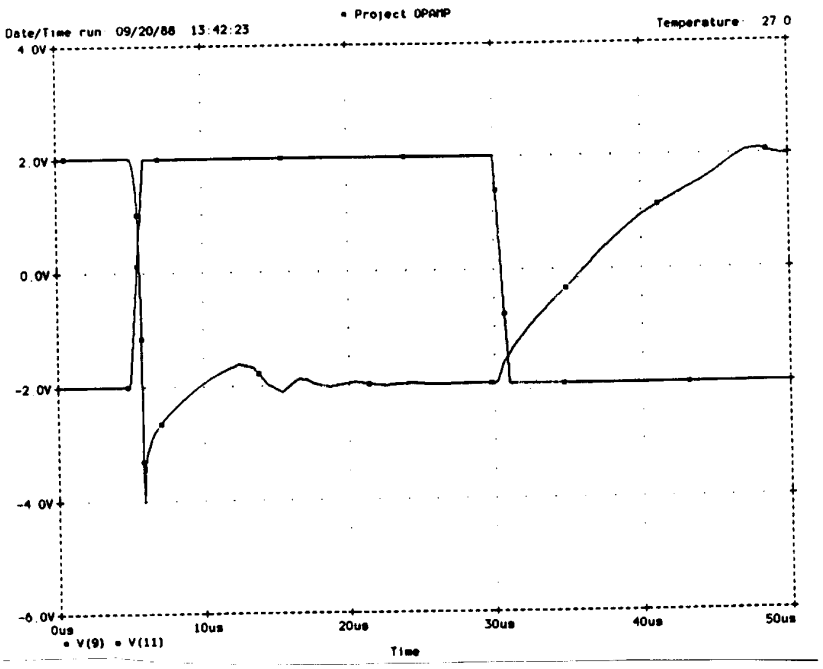

Figure 6.9. PROBE output of the analog inverter's transient response

Fig.6.9 shows the transient behavior of this opamp as displayed by the *Probe* feature. Signal $V(11)$ points to the input of the inverter while $V(9)$ shows the inverted output signal.

Obviously, this opamp is not all that great, but it suffices to illustrate the concept. After all, it is not the goal of this text to teach circuit design, but to teach modeling concepts. Fig.6.10 shows the AC-output (Bode diagram) of the opamp as shown by the *Probe* feature. $VDB(9)$ depicts the gain of the opamp in decibels, while $VP(9)$ shows its phase in degrees. The opamp has a band width of approximately 400 kHz.

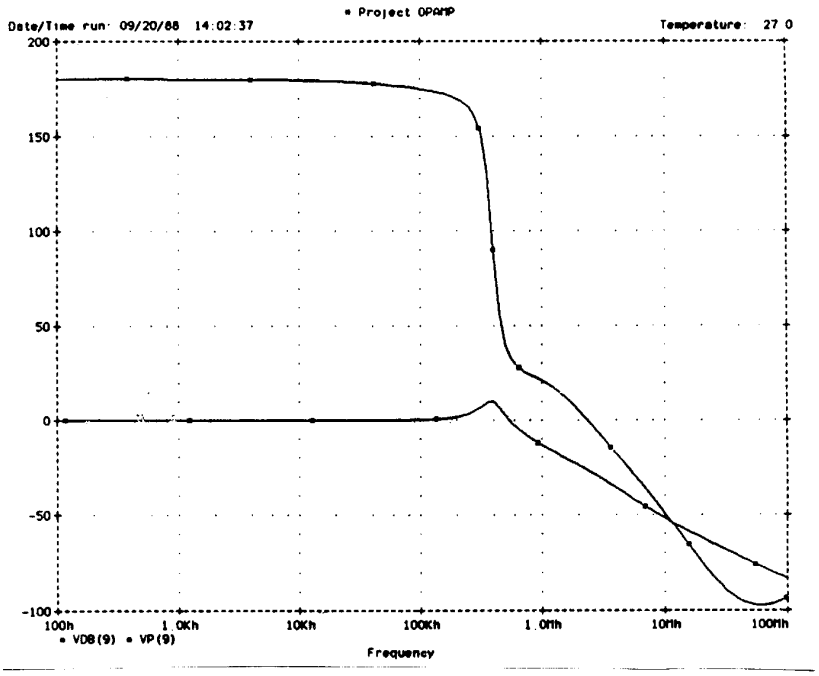

Figure 6.10. PROBE output of the analog inverter's AC response

6.6 Circuit Design Using DYMOLA

In Chapter 5, we have shown that DYMOLA might provide us with an alternative approach to handling complex systems. Let us see

how DYMOLA could be used as an alternative to SPICE for modeling electrical circuits. The advantages are obvious. If we are able to make DYMOLA powerful enough that it can handle arbitrarily complex circuits containing arbitrary algebraic loops and structural singularities, we can automatically generate a state–space model which will execute much more efficiently at run–time than the currently used SPICE code.

First, we need to model the basic circuit elements. This is straightforward. The following code shows the DYMOLA model types describing resistors, capacitors, inductors, voltage sources, and the ground. These models are equivalent to those used in BBSPICE.

```
model type resistor
  cut A(Va/I), B(Vb/-I)
  main cut C[A,B]
  main path P<A-B>
  local V, Rval
  parameter R=0.0, TR1=0.0, TR2=0.0, Area=1.0
  external DTemp, DTempSq
  V=Va-Vb
  Rval=(R+TR1*DTemp+TR2*DTempSq)/Area
  Rval*I=V
end

model type capacitor
  cut A(Va/I), B(Vb/-I)
  main cut C[A,B]
  main path P<A-B>
  local V, Cval
  parameter C=0.0, TC1=0.0, TC2=0.0, Area=1.0
  external DTemp, DTempSq
  V=Va-Vb
  Cval=(C+TC1*DTemp+TC2*DTempSq)*Area
  Cval*der(V)=I
end

model type inductor
  cut A(Va/I), B(Vb/-I)
  main cut C[A,B]
  main path P<A-B>
  local V, Lval
  parameter L=0.0, TL1=0.0, TL2=0.0
  external DTemp, DTempSq
  V=Va-Vb
  Lval=L+TL1*DTemp+TL2*DTempSq
  Lval*der(I)=V
end
```

```
model type vsource
  cut A(Va/I), B(Vb/ - I)
  main cut C[A, B]
  main path P < A - B >
  terminal V
  default V = 0.0
  V = Vb - Va
end

model type Common
  main cut A(V/.)
  V = 0.0
end
```

These DYMOLA models suffice to describe the simple passive circuit of Fig.6.1. The DYMOLA code needed to describe that circuit is as follows.

```
model RLC
  submodel(resistor)  R1, R2
  submodel(capacitor) C
  submodel(inductor)  L
  submodel(vsource)   U0
  submodel Common
  parameter Temp = 300.0
  constant TRoom = 300.0
  local DTemp, DTempSq
  internal DTemp, DTempSq
  input u
  output y
  DTemp = Temp - TRoom
  DTempSq = DTemp * DTemp
  connect Common - U0 - ((R1 - (C//R2))//L) - Common
  U0.V = u
  y = R2.Va
end
```

In this model, several new language elements were introduced. The "–" sign is equivalent to the keyword "**to**", and denotes a series connection, whereas the "//" symbol is equivalent to the keyword "**par**" and denotes a parallel connection.

It is now time to analyze DYMOLA's declaration statements a little more closely. In DYMOLA, all variables must be declared. *Constants* are variables that obtain once (upon declaration) a constant value, and that are never reassigned. *Parameters* are similar to constants. They never change their values during a simulation run.

They can be reassigned, but only between simulation runs. This enables the compiler to extract all parameter computations from the dynamic loop. Parameters are one mechanism for data exchange between models. In this context, parameters are similar to formal read–only arguments of a subprogram call in a traditional programming language. Parameters cannot be reassigned within the model in which they are declared as parameters, only within the calling program. *Externals* are similar to parameters, but they provide for an implicit rather than an explicit data exchange mechanism. In this respect, they are similar to COMMON variables in a Fortran program. Externals are used to simplify the utilization of global constants or global parameters such as the temperature *Temp* or the thermal voltage *VT*. For security reasons, the calling program must acknowledge its awareness of the existence of these globals, by specifying them as *internal*. Notice, however, that "internal" is not a declaration but only a provision for redundancy, i.e., all "internal" variables must be declared as something else also.

Variables that may change their values during a simulation run are either *locals* or *terminals*, depending on whether they are connected to the outside world, or whether they are local to the model in which they are used. Terminals can be assigned using a dot notation (as this was done in the case of the voltage source: $U0.V = u$), or they can form part of one or several *cuts* in which case they can be *connected*. *Inputs* and *outputs* are special types of terminals. Variables declared as "terminals" or in "cuts" are undirected variables. For example, the statement $V = Vb - Va$ of the model *vsource* can be rearranged by the compiler into either $Vb = V + Va$ or $Va = Vb - V$ if needed. Had V been declared as "output", the compiler would be prevented from rearranging this equation.

Of course, not all circuits are conveniently described by sets of series and parallel connections. Alternatively, DYMOLA allows us to formulate the circuit in a topological manner similar to that used in SPICE. The following code shows an alternate way to formulate the above circuit using DYMOLA.

```
model RLC
  submodel(resistor)  R1, R2
  submodel(capacitor) C
  submodel(inductor)  L
  submodel(vsource)   U0
  submodel Common
  node N0, N1, N2
  parameter Temp = 300.0
  constant TRoom = 300.0
  local DTemp, DTempSq
  internal DTemp, DTempSq
  input u
  output y

  DTemp = Temp - TRoom
  DTempSq = DTemp * DTemp

  connect - >
    Common at     N0, - >
    U0       from N0 to N1, - >
    R1       from N1 to N2, - >
    R2       from N2 to N0, - >
    C        from N2 to N0, - >
    L        from N1 to N0, - >

  U0.V = u
  y = R2.Va
end
```

The "- >" symbol denotes continuation lines in DYMOLA.

The generated code can be optimized by requesting DYMOLA to automatically throw out all terms that are multiplied by a parameter with value 0.0. Therefore, if the equation:

$$Cval = C + TC1 * DTemp + TC2 * DTemp * Dtemp \tag{6.16}$$

uses its default values of zero for both *TC1* and *TC2*, the equation would automatically degenerate into the equation

$$Cval = C \tag{6.17}$$

at which time another DYMOLA rule will be activated that will throw out this equation altogether, and replace the variable *Cval* by C in the subsequent equation

$$Cval * \mathbf{der}(V) = I \tag{6.18}$$

since parameters take preference over local variables. In this way, DYMOLA can achieve indirectly the same run–time savings as SPICE does since only significant equations and terms will survive the translation. DYMOLA can also be requested to automatically extract all parameter computations from the dynamic portion of the simulation code into the initial portion of the code.

Let us now investigate what it would take to simulate a BJT in DYMOLA. As shown in Fig.6.4, the BJT model contains three junctions, and thus, we require a junction diode model. Its code is presented below.

```
model type jdiode
  cut Anode(Va/I), Cathode(Vb/ − I)
  main cut C[Anode, Cathode]
  main path P < Anode − Cathode >
  local V, Ic, Qc, ISval, VDval, CDval
  terminal Id
  parameter ND = 1.0, IS = 1.0E-16, TD = 0.0, − >
            CD = 0.0, VD = 0.75, MD = 0.33, Area = 1.0
  external DTemp, FTemp, Gmin, XTI, − >
           VT, EGval, VDfact

  {Electrical equations}
  V = Va − Vb
  I = Id + Ic
  Id = ISval * (exp(V/(VT * ND)) − 1.0) + Gmin * V
  Ic = der(Qc)
  Qc = TD * Id − >
       +VDval * CDval * (1 − (1 − V/VDval) * *(1 − MD))/(1 − MD)

  {Temperature adjustment equations}
  ISval = IS * Area * exp((FTemp − 1.0) * EGval/VT) * FTemp * *XTI
  VDval = FTemp * (VD − VDfact) + VDfact
  CDval = CD * Area/(1.0 + MD * (1.0 − VDval/VD + 4.0E-4 * DTemp))
end
```

This model contains one algebraic loop which, however, can be solved easily. Obviously, we wish to solve the differential equation for the derivative. Therefore, Qc is a state variable. Thus, we must solve the equation $Qc = \ldots$ for another variable, namely either V or Id. However, the equation $Id = \ldots$ depends also on V, and thus, these two equations form an algebraic loop involving the variables V and Id. One way to solve this problem would be to replace the Id from the equation $Qc = \ldots$ by the other equation. In this way, the variable Id has been eliminated from the equation $Qc = \ldots$, and we can

thus solve this equation for V. Thereafter, we can use the equation $Id = \ldots$ to determine Id. Unfortunately, the resulting equation:

$$Q_c = \tau_d I_s[\exp(\frac{V}{V_T N_d}) - 1] + \tau_d G_{min} V + \frac{\phi_d C_d[1 - (1 - \frac{V}{\phi_d})^{1-m_d}]}{1 - m_d} \quad (6.19)$$

is highly non–linear in V, and does not have an analytical inverse. Thus, we must either simplify the equation until it *has* an analytical inverse, or employ a numerical iteration scheme to find V for any given value of Q_c from this equation.

The junction capacitance model shown above is not the one that is currently employed in BBSPICE since I recently implemented an improved model which had been proposed by Van Halen [6.10]. This model avoids, in an elegant fashion, the singularity that occurred in the previously used model at $V = \phi_d$. However, since the new equation is even more bulky than the one shown in eq(6.19), I shall refrain from presenting it here. Moreover, the inversion problem that was demonstrated by means of eq(6.19) remains exactly the same.

Next, we need a model for the variable base resistance of the BJT. Such a model is shown in the following code segment.

```
model type rbb
  cut A(Va/I), B(Vb/ - I)
  main cut C[A,B]
  main path P < A - B >
  local V, Rval, RBval, RBMval, z, tz
  constant pi = 3.14159
  parameter RB = 0.0, RBM = 0.0, IRB = 0.0, Area = 1.0, - >
            TRB1 = 0.0, TRB2 = 0.0, TRM1 = 0.0, TRM2 = 0.0
  external DTemp, DTempSq, qb

  V = Va - Vb
  RBval = (RB + TRB1 * DTemp + TRB2 * DTempSq)/Area
  RBMval = (RBM + TRM1 * DTemp + TRM2 * DTempSq)/Area
  Rval = if IRB = 0.0 - >
    then RBMval + (RBval - RBMval)/qb - >
    else RBMval + 3.0 * (RBval - RBMval) * (tz - z)/(z * tz * tz)
  z = if IRB = 0.0 - >
    then 0.0 - >
    else (-1 + sqrt(1 + 144 * I/(pi * pi * IRB * Area))) - >
         /(24 * sqrt(I/(IRB * Area))/(pi * pi))
  tz = if IRB = 0.0 then 0.0 else tan(z)
  Rval * I = V
end
```

This model demonstrates one of the weaknesses of DYMOLA. In order to be able to *sort* all equations properly, DYMOLA provides us with a funny looking "if" statement of the form:

$$< var >= \textbf{if} < cond > \textbf{then} < expr > \textbf{else} < expr >$$

This becomes quite awkward when a condition propagates through a number of statements. Notice that in this text, we shall usually present the full models only, and skip their degenerated versions in order to keep the models short and understandable. Notice further that also this model contains an algebraic loop for the case $IRB \neq 0.0$ involving the variables *Rval*, z, and I. The simplified model (for $IRB = 0.0$) does not contain any algebraic loop.

Next, we require a model for the non–linear external base junction capacitance *cbcx*. The following code describes this non–linear capacitance.

```
model type cbcx
  cut A(./I), B(./ - I)
  main cut C[A,B]
  main path P < A - B >
  local Qx, Vval, Cval
  terminal vbc
  parameter CJC = 0.0, MJC = 0.33, VJC = 0.75, XCJC = 1.0, - >
            Area = 1.0
  external DTemp, FTemp, VDfact

  Qx = Vval * Cval * (1 - XCJC) - >
       *(1 - (1 - vbc/Vval) * *(1 - MJC))/(1 - MJC)
  der(Qx) = I

  {Temperature adjustment equations}
  Vval = FTemp * (VJC - VDfact) + VDfact
  Cval = CJC * Area/(1.0 + MJC * (1.0 - VDval/VJC + 4.0E-4 * DTemp))
end
```

Finally, we need the BJT model itself. In SPICE, all four types (NPN and PNP, vertical and lateral) are coded in one single subroutine. In DYMOLA, this is hardly feasible at the moment. So, let us look at one of the transistor types only, namely the laterally diffused NPN transistor. The models for the other three types are similar.

```
model type NPNlat

 submodel (resistor) rcc(R = RC, TR1 = TRC1, TR2 = TRC2, ->
                         Area = Area), ->
                         ree(R = RE, TR1 = TRE1, TR2 = TRE2, ->
                         Area = Area)
 submodel rbb(RB = RB, RBM = RBM, IRB = IRB, Area = Area, ->
              TRB1 = TRB1, TRB2 = TRB2, TRM1 = TRM1, ->
              TRM2 = TRM2)
 submodel cbcx(CJC = CJC, MJC = MJC, VJC = VJC, ->
               XCJC = XCJC, Area = Area)
 submodel (jdiode) dbc(ND = NR, IS = IS, TD = TR, Area = Area, ->
                       CD = CJC * XCJC, VD = VJC, MD = MJC), ->
                   dbe(ND = NF, IS = IS, TD = TF, Area = Area, ->
                       CD = CJE, VD = VJE, MD = MJE), ->
                   dbs(ND = NS, IS = ISS, TD = 0.0, Area = Area, ->
                       CD = CJS, VD = VJS, MD = MJS)
 submodel (csource) ice0, ibe0
 cut Collector(VC/IC)
 cut Base(VB/IB)
 cut Emitter(VE/ - IE)
 cut Substrate(VS/ISUB)
 main cut CBES [Collector, Base, Emitter, Substrate]
 path Basemitter < Base - Emitter >
 path Colemitter < Collector - Emitter >
 node IntCollector, IntBase, IntEmitter

 constant pi = 3.14159
 parameter IS = 1.0E-16, ISC = 0.0, ISE = 0.0, ISS = 0.0, ->
           BF = 100.0, BR = 1.0, TF = 0.0, TR = 0.0 ->
           NC = 2.0, NE = 1.5, NF = 1.0, NR = 1.0, NS = 1.0, ->
           VAF = 0.0, VAR = 0.0, IKF = 0.0, IKR = 0.0, ->
           VJC = 0.75, VJE = 0.75, VJS = 0.75, ->
           CJC = 0.0, CJE = 0.0, CJS = 0.0, ->
           MJC = 0.33, MJE = 0.33, MJS = 0.5, ->
           RB = 0.0, RBM = 0.0, RC = 0.0, RE = 0.0, ->
           TRB1 = 0.0, TRM1 = 0.0, TRC1 = 0.0, TRE1 = 0.0, ->
           TRB2 = 0.0, TRM2 = 0.0, TRC2 = 0.0, TRE2 = 0.0, ->
           XCJC = 1.0, EG = 1.11, XTI = 3.0, IRB = 0.0, Area = 1.0
 local q1, q2, qb, xti, vbc, vbe, ibc, ibe, ibcn, ->
       iben, IB0, IC0, EGval, EGroom, VDfact
 external Temp, TRoom, DTemp, DTempSq, FTemp, ->
          Charge, Boltz, VT, VTroom, BT, BTroom, ->
          GapC1, GapC2, Gmin
 internal DTemp, DTempSq, FTemp, VT, qb, EGval, ->
          VDfact, Gmin, xti
```

```
{Compute frequently used internal voltages and currents}
vbc = rbb.Vb - rcc.Vb
vbe = rbb.Vb - ree.Va
ibc = dbc.Id
ibe = dbe.Id

{Compute the base charge}
q1 = 1.0/(1.0 - vbc/VAF - vbe/VAR)
q2 = (ibc/IKR + ibe/IKF)/Area
qb = q1 * (1.0 + sqrt(1.0 + 4.0 * q2))/2.0
xti = XTI

{Compute the nonlinear current sources}
ibcn = ISC * Area * (exp(vbc/(VT * NC)) - 1.0)
iben = ISE * Area * (exp(vbe/(VT * NE)) - 1.0)
IC0 = (ibe - ibc)/qb - ibc/BR - ibcn
IB0 = ibe/BF + iben + ibc/BR + ibcn - ibe/qb

{Compute the globals}
EGroom = EG - GapC1 * TRoom * *2/(TRoom + GapC2)
EGval = EG - GapC1 * Temp * *2/(Temp + GapC2)
VDfact = -2.0 * VT * (1.5 * log(FTemp) - >
         -0.5 * Charge * (EGval/BT - EGroom/BTroom))

{Plug the internal circuit together}
connect rbb  from Base        to IntBase
connect rcc  from Collector   to IntCollector
connect ree  from IntEmitter  to Emitter
connect dbc  from IntBase     to IntCollector
connect dbe  from IntBase     to IntEmitter
connect dbs  from IntBase     to Substrate
connect cbcx from Base        to IntCollector
connect ibe0 from IntBase     to IntEmitter
connect ice0 from IntCollector to IntEmitter

ice0.I0 = IC0
ibe0.I0 = IB0
cbcx.vbc = vbc

end
```

This model is currently not without its problems. It seems that the BJT is described as a fourth order model since it contains four separate capacitances each of which is described through a first order differential equation. Unfortunately, this is an example of a degenerate system, as can be easily shown. Let us assume that we describe the capacitance of the base–collector junction diode through a differential equation. Then, the charge over this capacitance $dbc.Qc$ is

a state variable, and we need to solve the "*dbc.Qc* = ..." equation for the variable *dbc.V*. However, this variable is the same variable as *NPNlat.vbc* which is connected to the *terminal* variable of the external base collector junction capacitance, i.e., *cbcx.vbc* = *NPNlat.vbc* = *dbc.V*. Therefore, we must solve the charge equation of the external capacitance for *cbcx.Qx* which gets us into trouble since we would also like to solve this variable from the differential equation **der**(*cbcx.Qx*) = *cbcx.I*.

In order to overcome this model degeneracy, we would need to analytically compute the derivative of the equation *cbcx.Qx* = ..., and eliminate the differential equation.

Currently, the major benefit of these DYMOLA circuit descriptions is with their *documentary value*. We feel that our BJT model description is much more readable and understandable than the description given in most SPICE manuals, and it is also much easier to read than the SPICE source code listing. Currently, several of our graduate students are working on DYMOLA to make the software more powerful than it currently is, and one of them is explicitly looking at DYMOLA as a tool for electrical circuit design.

6.7 How DYMOLA Works

Until now, we have only discussed how an input file for DYMOLA (i.e., a hierarchical model) is to be prepared. We have not yet seen what the DYMOLA preprocessor does with this model upon execution. This will be demonstrated now by means of a very simple example. Fig.6.11 shows an almost trivial electrical circuit consisting of one voltage source and two resistors.

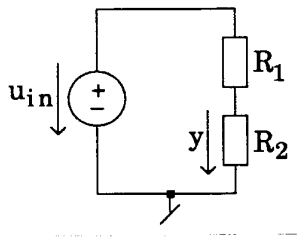

Figure 6.11. Schematic of a trivial electrical circuit

Using the electrical component models that were introduced earlier in this chapter, the above circuit can be described through the following DYMOLA program.

```
@vsource.elc
@resistor.elc
@common.elc

model circuit
  submodel (vsource) Uin
  submodel (resistor) R1(R = 10.0), R2(R = 20.0)
  submodel Common
  input u
  output y
  connect Common – Uin – R1 – R2 – Common
Uin.V0 = u
y = R1.Vb
end
```

The @ operator instructs the DYMOLA preprocessor to include an external file at this place. A DYMOLA program may contain definitions for (or inclusions of) an arbitrary number of *model types* followed by exactly one *model* which invokes the declared model types as *submodels.*

The following command sequence calls upon the DYMOLA preprocessor which is requested immediately to read in the model definition:

```
$ dymola
  > enter model
  – @circuit.dym
  >
```

At the operating system prompt ($), we call the DYMOLA preprocessor which enters into an interactive mode, and responds with its own prompt (>). The next statement instructs DYMOLA to read in a model. DYMOLA will present us with the next level prompt (–), until it has read in a complete model specification. We could enter equations here, but it is more practical to invoke them indirectly (@). At this point, DYMOLA is satisfied since it found a model definition, and returns to its first level interactive prompt (>).

Already at this point, DYMOLA has replaced all *submodel* references by their model definitions, and has generated the additional equations that are a result of the submodel couplings (i.e., DYMOLA has replaced the *connect* statements by the coupling equations. The

result of this text replacement can be observed by issuing the command:

```
> output equations
```

which will result in the following display:

Common	$V = 0.0$
Uin	$V0 = Vb - Va$
R1	$V = Va - Vb$
	$R * I = V$
R2	$V = Va - Vb$
	$R * I = V$
circuit	$Uin.V0 = u$
	$y = R1.Vb$
	$R1.Va = Uin.Vb$
	$R1.I = Uin.I$
	$R2.Va = R1.Vb$
	$R2.I = R1.I$
	$R2.Vb = Uin.Va$
	$Common.V = R2.Vb$

assuming that a simplified resistor model was used which does not include the temperature variation effects and the area parameter. The output can be redirected to a file (for printout) using the command:

```
> outfile circuit.eq1
```

to be issued prior to the *output equations* command.

At this point, we can try to determine which equation needs to be solved for what variable, and simultaneously, sort the equations into an executable sequence. This algorithm was thoroughly described in Chapter 3. In DYMOLA, the algorithm is invoked by issuing a *partition* command:

```
> partition
```

Thereafter, we may wish to look at the marked and sorted but not yet solved equations. This is achieved with the command:

```
> outfile circuit.sr1
> output sorted equations
```

which will write the following text to the file *circuit.sr1*:

Common	$[V] = 0.0$
circuit	$Common.V = [R2.Vb]$
	$R2.Vb = [Uin.Va]$
	$[Uin.V0] = u$
Uin	$V0 = [Vb] - Va$
circuit	$[R1.Va] = Uin.Vb$
–	$R2.I = [R1.I]$
$-R2$	$R * [I] = V$
–	$[V] = Va - Vb$
$-circuit$	$[R2.Va] = R1.Vb$
$-R1$	$V = Va - [Vb]$
–	$R * I = [V]$
circuit	$R1.I = [Uin.I]$
	$[y] = R1.Vb$

A first set of six equations was sorted correctly. This is followed by another set of six equations that form a *linear algebraic loop.* DYMOLA marks the equations belonging to an algebraic loop with $-$. Finally, the last two equations can be properly sorted once the algebraic loop is solved. As this example demonstrates, algebraic loops occur indeed rather commonly.

We can now proceed to *optimize* the code. The above set of equations contains many *aliases*, i.e., the same physical quantity is stored several times under different variable names. This will slow down the execution of our simulation program. The command:

> *eliminate equations*

gets rid of equations of the type:

$$a = b$$

and replaces all occurrences of the variable a in all other equations by the symbol b. One exception to the rule must be stated: The eliminate operation never eliminates a variable that was declared as either *input* or *output.* If a is an output variable, DYMOLA will throw the equation away as well, but in this case, all occurrences of b are replaced by a. If both a and b are declared as *output* variables, the equation will not be eliminated at all.

This algorithm can be applied either to the original equations or to the partitioned equations, and it will work equally well in both cases. The resulting code is shown below:

Common	$[Uin.Va] = 0.0$
Uin	$circuit.u = [Vb] - Va$
$-R2$	$R * [Uin.I] = V$
–	$[V] = circuit.y - Uin.Va$
$-R1$	$V = Uin.Vb - [circuit.y]$
–	$R * Uin.I = [V]$

In reality, this algorithm reduces all equations of the types:

$$\pm a = \pm b$$

and:

$$\pm a \pm b = 0$$

which are variants of the previously discussed case. The algorithm works also if either a or b is a constant. Consequently, the above set of equations is not yet the "final product". Below, the truly reduced set of equations is presented.

$-R2$	$R * [Uin.I] = circuit.y$
$-R1$	$V = circuit.u - [circuit.y]$
–	$R * Uin.I = [V]$

We can now optimize the code a little further by requesting:

> *eliminate parameters*

The algorithm that is now executed will perform the following tasks:

(1) All parameters with a numerical value value of 0.0 or 1.0 are eliminated from the model, and the numerical value is replaced directly into the equations.

(2) A numerical value of 1.0 that multiplies a term is eliminated from that term.

(3) A term that is multiplied by a numerical value of 0.0 is replaced as a whole by 0.0.

(4) Additive terms of 0.0 are eliminated altogether.

(5) If, in an equation, an expression consists of parameters and constants only, a new equation is generated that will evaluate this expression (assigned to a new generic variable), and the occurrence of the expression in the equation is replaced by the new generic variable.

(6) If an equation contains only one variable, it must be solved for that variable. This variable is then automatically redeclared as a parameter, and the equation is marked as a *parameter equation* which can be moved from the DYNAMIC portion of the simulation program into the INITIAL portion of the simulation program.

In our example, nothing will happen if we apply this algorithm since none of the parameters has a value of either 1.0 or 0.0. However, in reality, we did not use a simplified resistor model, but we used the full resistor model in which the temperature variation coefficients defaulted to 0.0, and in which the *Area* parameter defaulted to 1.0. Applying the above algorithm to the full resistor model will reduce the equations to exactly the format that we found in our last display.

Notice that this algorithm may have undesirable side effects. Often, we may wish to start off with a simple model (by setting some parameters equal to 0.0), and then successively make the model more realistic by assigning, in the simulation program, true values to the previously defaulted parameters. In this case, we shouldn't *eliminate parameters* since this algorithm will put the eliminated parameters to the sword once and for all. These parameters will no longer appear in the generated simulation program.

We can now proceed to *eliminate variables.* This algorithm can only be applied after the equations have been partitioned. It affects only algebraic loops, and it affects each algebraic loop in the model separately. DYMOLA counts the times that each loop variable is referenced in an algebraic loop. Obviously, each loop variable must occur at least twice, otherwise, it would not be a loop variable. If we request DYMOLA to:

```
> eliminate variables
```

then DYMOLA will investigate all loop variables that occur exactly twice in a loop. If it found such a variable, and if this variable appears linearly in at least one of the two equations, DYMOLA will solve the equation for that variable, and replace the other occurrence of the variable by the evaluated expression, thereby eliminating this variable altogether from the loop. If the eliminated variable is referenced anywhere *after* the loop, the equation defining this variable is not thrown away, but taken out of the loop and placed immediately after the loop. The same is true if the eliminated variable has been declared as an *output* variable. Although the same algorithm could be applied to variables that occur more than twice, this is not done since the algorithm tends to expand the code (the same, possibly long, expressions are duplicated several times).

If we apply this algorithm to our example, we first notice that our algebraic loop contains the three loop variables *Uin.I*, *circuit.y*, and *R1.V*. Each of them appears exactly twice in the loop. We investigate *Uin.I* first, and find that it occurs linearly in the first equation. Therefore, we solve that equation for *Uin.I*, replace the found expression in the third equation, and eliminate the variable since it was not declared as an output variable. The result of this operation is as follows:

$$\begin{array}{ll} -R1 & V = circuit.u - [circuit.y] \\ - & R * (circuit.y/R2.R) = [V] \end{array}$$

Since *circuit.y* is an output variable, we prefer to analyze $R1.V$ next. We find that it occurs linearly in the second equation, solve for it, and replace the result in the first. The result is as follows:

$$R1 \qquad R * ([circuit.y]/R2.R) = circuit.u - [circuit.y]$$

at which time the algebraic loop has disappeared. DYMOLA's equation solver is able to turn this equation into:

$$R1 \qquad circuit.y = circuit.u/(1 + R/R2.R)$$

At this point, we can once more *eliminate parameters*, which will lead to the following code:

Initial :
$D00001 = 1/(1 + R1.R/R2.R)$
Dynamic :
$circuit.y = D00001 * circuit.u$

DYMOLA offers yet another elimination algorithm. If we request DYMOLA to:

> *eliminate outputs*

DYMOLA will check for all variables in the DYNAMIC section of the code that appear only once in the set of equations. Obviously, the equations containing these variables must be used to evaluate them. Since these equations will not otherwise influence the behavior of the dynamic model, they can be marked as *output equations*, and can be taken out of the state–space model. If we apply this algorithm to our example, the following code results:

Initial :
$D00001 = 1/(1 + R1.R/R2.R)$
Dynamic :
Output :
$circuit.y = D00001 * circuit.u$

The command:

```
> partition eliminate
```

will partition the equations, and then automatically perform *all* types of elimination algorithms repetitively until the equations no longer change.

At the time of writing, not all of the above code optimization techniques have been fully implemented yet. However, the command *partition eliminate* can already be used. It simply skips those optimization tests that haven't been fully implemented.

We can receive a printout of the *solved equations* by issuing the command sequence:

```
> outfile circuit.sv1
> output solved equations
```

At this point, the circuit topology has been reduced to a (in this case trivial) state–space model. Now, DYMOLA's code generator portion can be used to generate a simulation program either for DESIRE [6.5], SIMNON [6.2], or FORTRAN. However, before we can do so, we need to add an experiment description to our DYMOLA program. The experiment may look as follows:

```
cmodel
   simutime 2E-5
   step 2E-7
   commupoints 101
   input 1, u(independ, 10.0)

ctblock
   scale = 1
   XCCC = 1
   label TRY
   drunr
if XCCC < 0 then XCCC = -XCCC | scale = 2 * scale | go to TRY
            else proceed
ctend

outblock
   OUT
   dispt y1, y2
outend
end
```

This portion of code is specific for each of the target languages. The version shown here is the one required for DESIRE [6.5]. The

ctblock set of statements instructs DESIRE to automatically scale the run–time display. *XCCC* is a DESIRE variable which is set to -1 whenever the DESIRE program is interrupted with an "overflow". This happens when one of the displayed variables hits either the top or the bottom of the displayed window. At this time, the plot is simply rescaled, and the simulation is rerun with a new *drunr* statement. Since DESIRE is so fast, it is not worth the effort to store the results of the previous attempt, instead, we simply rerun the entire simulation.

The experiment description is entered into DYMOLA using the statements:

```
> enter experiment
  – @circuit.ctl
>
```

Now, we are ready to generate the simulation program:

```
> outfile circuit.des
> output desire program
```

To exit from DYMOLA and enter the DESIRE [6.5] program, we issue the command sequence:

```
  > stop
$ desire 0
  > load 'circuit.des'
  > save
  > run
  > bye
$
```

The 0 parameter on the DESIRE call instructs DESIRE to not automatically load the last executed DESIRE program into memory, the *load* command loads the newly generated DESIRE program into memory, the *save* command saves the program in binary form onto the file *circuit.prc* for a quicker reload at a later time using the command:

```
> old 'circuit'
```

The *run* command finally executes the DESIRE program, and the *bye* command returns control to the operating system.

6.8 Summary

In this chapter, we have discussed the tools that commercial circuit designers use to analyze and design their circuits. PSpice [6.9] and BBSPICE [6.1] are two of several available SPICE and ASTAP dialects. Workview [6.11] is one of a number of available schematic capture programs. I chose to mention those programs in the text, since I have the most familiarity with them. However, I truly believe that, among the programs that I have had a chance to review, the selected ones are indeed the most powerful. With respect to circuit analysis programs, PSpice has probably today become the most popular among the SPICE dialects due to its availability and excellent implementation on PC–class machines. With respect to the schematic capture programs, some of the other commercially available programs don't provide an interface to PSpice, or their interface (the so–called netlisting feature) executes much slower, or they run only on more specialized hardware (such as Apollo workstations), hardware that is not as readily available, or they are considerably more expensive. Finally, Workview offers a very convenient feature related to the simulation of digital circuits. It contains a *logic simulator* which allows the user to quickly analyze the behavior of a digital circuit before s/he ever goes through the slow and painful process of decomposing the circuit down to the transistor level, and simulating it in greater detail using PSpice.

We then returned to DYMOLA [6.3], and discussed the potential of this powerful modeling tool for circuit modeling. While DYMOLA is not yet capable of dealing with complex electronic circuitry due to the frequently cited problems with algebraic loops and structural singularities, I am convinced that eventually DYMOLA can become a true alternative to SPICE.

We shall return once more to electronic circuit simulation in the second volume of this text to discuss, in more detail, how the numerical integration (or rather differentiation) is performed in SPICE.

The aim of this chapter was to discuss circuit analysis from a modeling and simulation perspective, and not from the perspective of transistor circuit design. Dozens of texts are on the market which discuss transistor circuits very elegantly and in great detail. We didn't see a need to duplicate these efforts here. The bibliography of this chapter lists a more or less arbitrary collection of textbooks that deal with electronic circuit design issues.

References

[6.1] Burr Brown Corp. (1987), *BBSPICE — User's Manual*, 6730 S. Tucson Blvd., Tucson, AZ 85706.

[6.2] Hilding Elmqvist (1975), *SIMNON - An Interactive Simulation Program for Non-linear Systems — User's Manual*, Report CODEN: LUTFD2/(TFRT–7502), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[6.3] Hilding Elmqvist (1978), *A Structured Model Language for Large Continuous Systems*, Ph.D. Thesis, Report CODEN: LUTFD2/(TRFT–1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[6.4] Hewlett–Packard Company (1988), *HP 94430A/94431A Tecap User's Manual, HP 94431A Tecap Device Modeling and Parameter Extraction Manual*, 5301 Stevens Creek Blvd., Santa Clara, CA 95052.

[6.5] Granino A. Korn (1989), *Interactive Dynamic–System Simulation*, McGraw–Hill, New York.

[6.6] Mentor Corp. (1987), *ACCUSIM User's Manual*, Palo Alto, CA 94304.

[6.7] Mentor Corp. (1987), *ACCULIB User's Manual*, Palo Alto, CA 94304.

[6.8] Meta–Software, Inc. (1985), *HSPICE User's Manual*, 50 Curtner Ave., Suite 16, Campbell, CA 95008.

[6.9] MicroSim Corp. (1987), *PSpice User's Manual*, 20 Fairbanks Rd., Irvine, CA 92718.

[6.10] Paul Van Halen (1988), "A New Semiconductor Junction Diode Space Charge Layer Capacitance Model", *Proceedings IEEE 1988 Bipolar Circuits & Technology Meeting* (Janice Jopke, ed.), Minneapolis, Minn., IEEE Publishing Services, 345 47^{th} Street, New York, NY 10017, pp. 168–171.

[6.11] Viewlogic Systems, Inc. (1988) *Workview Reference Guide*, Release 3.0, *Viewdraw Reference Guide*, Version 3.0, 313 Boston Post Rd. West, Marlboro, MA 01752.

Bibliography

[B6.1] Alfred W. Barber (1984), *Practical Guide to Digital Electronic Circuits*, Second Edition, Prentice–Hall, Englewood Cliffs, N.J.

[B6.2] David J. Corner (1981), *Electronic Design with Integrated Circuits*, Addison–Wesley, Reading, MA.

[B6.3] Sergio Franco (1988), *Design with Operational Amplifiers and Analog Integrated Circuits*, McGraw–Hill, New York, N.Y.

[B6.4] Mohammed S. Ghausi (1984), *Electronic Devices and Circuits: Discrete and Integrated*, Holt, Rinehart, and Winston, New York, N.Y.

[B6.5] Jerald G. Graeme (1977), *Designing with Operational Amplifiers: Applications Alternatives*, McGraw–Hill, New York, N.Y.

[B6.6] Adel S. Sedra, and Kenneth C. Smith (1982), *Microelectronic Circuits*, Holt, Rinehart, and Winston, New York, N.Y.

Homework Problems

[H6.1] Tunnel Diode

A forward biased tunnel diode can be represented through the static characteristic shown in Table H6.1:

Table H6.1 Tunnel diode characteristic

voltage u_d $[V]$	current i_d $[mA]$
0.00	0.00
0.05	1.70
0.10	2.90
0.15	4.00
0.20	4.75
0.25	5.00
0.30	4.90
0.35	4.25
0.40	3.20
0.45	2.35
0.50	2.00
0.55	2.20
0.60	2.70
0.65	3.50
0.70	4.35
0.75	5.30
0.90	9.00

Use CTRL–C (or MATLAB) to obtain a graphical representation of this static characteristic.

We wish to analyze the behavior of the circuit shown in Fig.H6.1a.

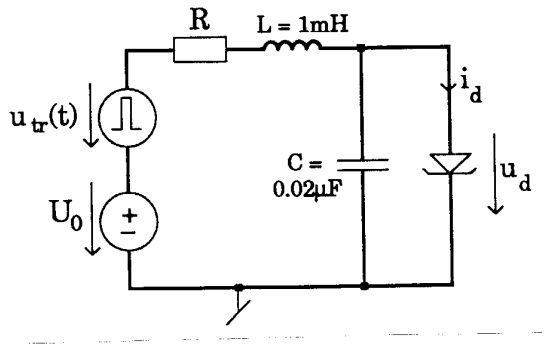

Figure H6.1a. Schematic of a circuit with a tunnel diode

In the first configuration, we shall use a resistor of $R = 25\ \Omega$, a DC–bias of $U_0 = 0.48\ V$, and a driver voltage of $u_{tr}(t) = 0.0\ V$, i.e., we analyze the behavior of this circuit under "free–running" conditions. Write a simulation program in ACSL that implements this circuit, simulate the circuit during 0.2 *msec*, and display the diode voltage u_d and the diode current i_d as functions of time. Describe the behavior of this circuit in qualitative terms.

In a second experiment, we wish to analyze the same circuit under modified experimental conditions. This time, we shall use a resistor of $R = 200\ \Omega$, and a DC–bias of $U_0 = 1.075\ V$. The driver voltage is a pulsed voltage source as depicted in Fig.H6.1b:

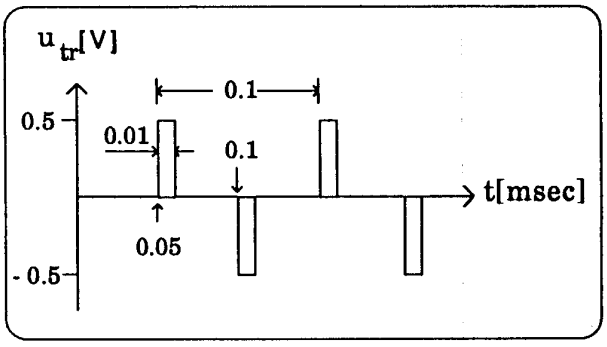

Figure H6.1b. Pulsed driver voltage source

Implement this modified setup in ACSL using the time–event scheduling facility to describe the pulsed driver voltage. The first two time–events are scheduled in the *INITIAL* section of the program, namely a time–event by the name of *VON* to occur at $t = 0.05$ *msec*, and another time–event by the name of *VOFF* to occur at time $t = 0.1$ *msec*. As part of each event description, a new event of the same type is scheduled to occur $\Delta t = 0.05$ *msec* later.

[H6.2] Differential Equations

We want to analyze the step response of a system described by the following transfer function:

$$G(s) = \frac{10}{s^2 + 2s + 10} \qquad (H6.2)$$

In a first experiment, use CTRL–C (MATLAB) to transform this frequency domain representation into a state–space representation, and simulate the step response of this system during 10 *sec* directly in CTRL–C (MATLAB).

As an alternative, design a simple passive circuit that could be used to obtain the same step response. Write a PSpice program that implements this circuit, perform a transient analysis, and display the obtained step response with the *Probe* option.

Since we don't want SPICE to perform a DC–analysis first in this case, we can prevent this from happening by adding the *UIC* qualifier to our transient analysis command:

```
.TRAN  0.1  10.0  UIC
```

[H6.3] Analog Computers

Analog computers are electronic circuits that basically consist of three different element types: *analog adders*, *analog integrators*, and *potentiometers*. We have already met the analog adder (cf. Fig.6.8). Adders used in commercial analog computers usually provide several inputs with a gain factor of -1.0 (i.e., 1 V at the input results in -1 V at the output), and one or several inputs with a gain factor of -10.0. A gain factor of -10.0 is achieved by making the corresponding input resistor 10 times smaller than the feedback resistor. The operational amplifier can ideally be represented through a high gain amplifier with an amplification of $A = -1.35 \times 10^8$, for example. An analog integrator can be built in almost the same way, just by replacing the feedback resistor of $R_f = 1\ M\Omega$ with a feedback capacitor of $C_f = 1\ \mu F$. In this circuit, 1 V at a unity gain input results in a ramp at output which, after 1 *sec* reaches a value of -1 V (assuming zero initial

conditions). When the 1 V input is applied to an input node with a gain factor of -10.0, the ramp reaches -1 V already after 0.1 *sec*.

Build a PSpice subcircuit for an ideal high gain amplifier. This can be achieved using a voltage driven voltage source (an E–element). Then build subcircuits for the analog adder and the analog integrator using the previously designed ideal high gain amplifier as a component. Finally, build a subcircuit for the potentiometer. This can again be achieved with a voltage driven voltage source, but this time, the gain A is variable with allowed values anywhere between 0.0 and 1.0. Depending on the SPICE dialect that you use, you may be able to specify the gain A as a parameter of the subcircuit, but PSpice does not support this feature. If your SPICE version does not allow you to specify parameters for your subcircuits, it may not be worthwhile to specify the potentiometer as a subcircuit at all.

Draw a block diagram for the state–space model of the differential equation of hw(H6.2), and convert it to an analog computer program. Don't forget that both the adder and the integrator contain built in inverters. The traditional symbols (icons) used to denote analog computer components are shown in Fig.H6.3.

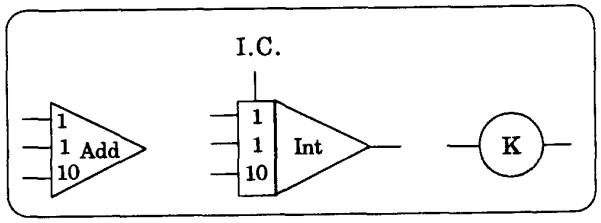

Figure H6.3. Components of analog computers

Use PSpice to simulate the just derived analog computer circuit. Since you don't want SPICE to perform a DC–analysis first in this case, you can prevent this from happening by adding the *UIC* qualifier to your transient analysis command:

.TRAN 0.1 10.0 *UIC*

It would now be possible to replace the ideal high gain amplifier by the operational amplifier subcircuit presented earlier in this chapter. However, we won't do this since the simulation will take forever. The time step of the transient analysis (the numerical integration) would have to be adjusted to the (very fast) time constants inside the BJT's which are in the order of 100 *nsec*, and a simulation of the overall circuit during 10 *sec* would thus not be realistic. We ran this model for roughly 1 *hour* on a VAX–3600 (using BBSPICE), and were able to complete (correctly) just the first

1 *msec* of the simulation. Consequently, the total simulation would require about 10000 *hours*. It was explained previously that modeling must be always goal driven. A finer granularity of the model does not necessarily make the model any better. Quite the contrary can be true as this example demonstrates.

[H6.4] Logic Gates

Fig.H6.4 shows a circuit for an OR/NOR gate in ECL logic. In this type of logic, $-1.8\ V$ corresponds to a logical *false* (or *off*), whereas $-0.8\ V$ corresponds to a logical *true* (or *on*).

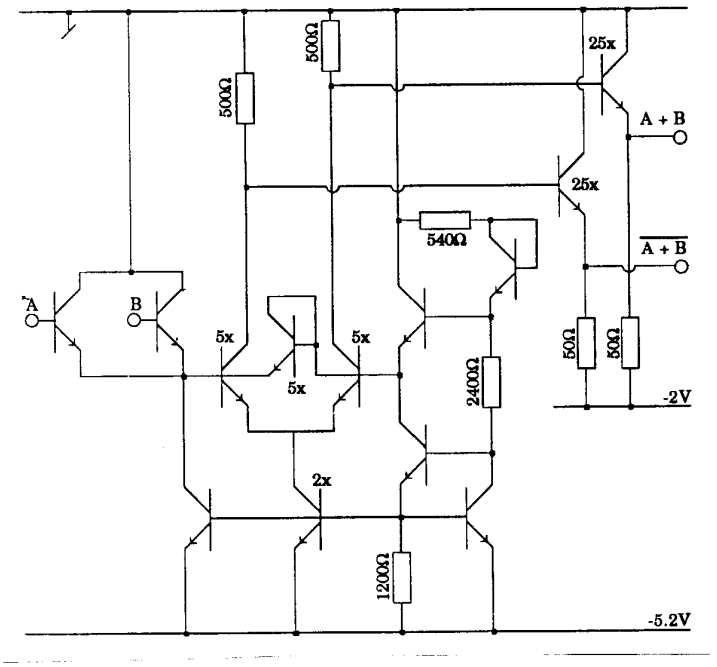

Figure H6.4. Circuit for an OR/NOR gate in ECL logic

Build a PSpice program that simulates the above circuit during 25 μsec. Use our *PROC35.N* model to describe the *NPN* transistors. Connect the substrates of all transistors to the lowest voltage, i.e. to $-5.2\ V$. The *Area* parameter to be used for the transistors is indicated on the schematic except for those transistors that use the default value of 1.0.

Keep the input A at $-1.8\ V$ during the first 5 μsec, then raise it to $-0.8\ V$, and keep it at that level until $t = 15\ \mu sec$. Thereafter, the input

A is kept at $-1.8\ V$ for the rest of the simulation. The second input, B, behaves similarly, except that it is raised from $-1.8\ V$ to $-0.8\ V$ at $t = 10\ \mu sec$, and it is reset to $-1.8\ V$ at $t = 20\ \mu sec$. Use the *PULSE* function to model the two time dependent voltage sources. Choose a raise and drop time for the input pulses of 100 *nsec* each.

Probe several of the voltages and currents in the circuit to come up with an explanation as to how this circuit works.

Reduce the raise and drop times of the two inputs until they are clearly faster than those produced by the circuit. Zoom in on the raising and dropping edges of the two output signals, and determine the natural raise and drop times of the the two output signals.

[H6.5] Digital to Analog Converter (DAC)

Fig.H6.5 shows a four bit digital to analog converter that can be driven by ECL logic.

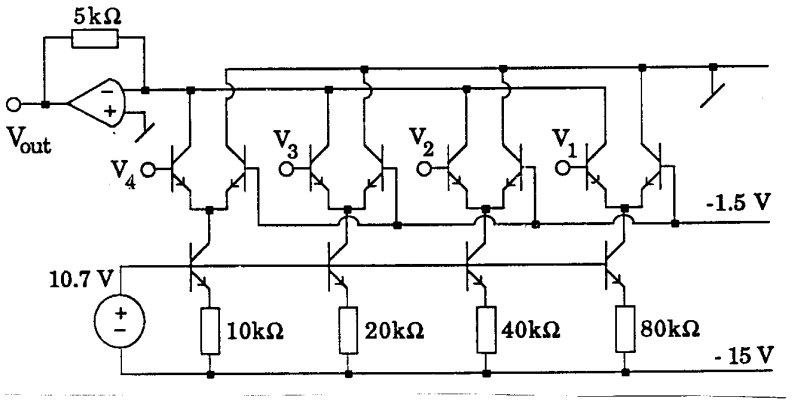

Figure H6.5. Four bit DAC for ECL logic

V_1 represents the least significant bit (LSB), while V_4 represents the most significant bit. If your version of SPICE supports subcircuits with formal parameters, design a subcircuit that describes one of the four DAC stages, and call it four times with different values of the parameter R. If your SPICE version does not support parameters, you better leave the resistor out of the subcircuit, and place it in the main program. Use the ideal opamp of hw(H6.3) for the analog output stage. Connect the substrates of all BJT's to the lowest voltage in the circuit, i.e., to $-15\ V$. Apply the following voltages to the four input nodes of the circuit:

```
V1 1 0 PULSE(-1.8 -0.8 24U 100N 100N 23.8U 48U)
V2 2 0 PULSE(-1.8 -0.8 12U 100N 100N 11.8U 24U)
V3 3 0 PULSE(-1.8 -0.8  6U 100N 100N  5.8U 12U)
V4 4 0 PULSE(-1.8 -0.8  3U 100N 100N  2.8U  6U)
```

and simulate the circuit during 60 μsec. *Probe* several of the voltages and currents in the circuit to understand how the circuit works. Explain the behavior of the output signal $V_{out}(t)$.

This time, we want to replace the ideal opamp by the transistorized opamp described earlier in this chapter. Tie the positive supply of the opamp, V_{CC} to $+15\ V$, and the negative supply, V_{EE} to $-15\ V$.

Repeat the previous experiment. You may have difficulties with DC–convergence. Possibly, you may have to ramp up all the supplies during 2 μsec and let them settle down during another 2 μsec, in order to obtain a decent initial nodeset. In that case, don't forget to switch all active devices (BJT's) *off* for the ramping experiment. Use the resulting OP–point as a nodeset for the subsequent experiment in which you perform the desired transient analysis.

[H6.6] Circuit Modeling in DYMOLA

In Fig.H6.6, a simple passive circuit is presented. The only new component is a current source for which a DYMOLA model type *csource* must be derived in analogy to the voltage source *vsource*.

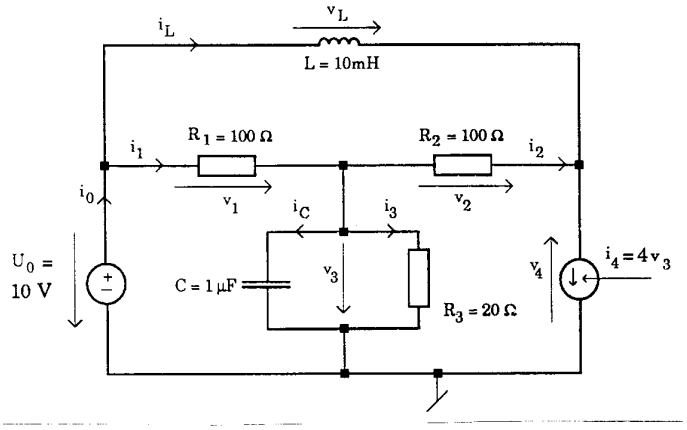

Figure H6.6. Circuit diagram of a simple passive circuit

Augment DYMOLA's electrical component library by a model type describing the current source. The fact that the current source in our example is

a dependent current source does not make any difference here. Contrary to SPICE, DYMOLA does not force us to distinguish between independent and dependent sources.

Derive a model for the overall circuit, and partition the equations once using the *partition* command and once using the *partition eliminate* command. In each of these cases, print out the *equations*, the *sorted equations*, and finally the *solved equations*. Thereafter, generate either a SIMNON [6.2] or a DESIRE [6.5] program, and simulate the system over 50 μsec. Use a step size of 50 *nsec*. Compare the results with those of hw(H3.4).

[H6.7]* Logic Inverter Modeled in DYMOLA

Fig.H6.7 shows the schematic of a simple logic inverter model.

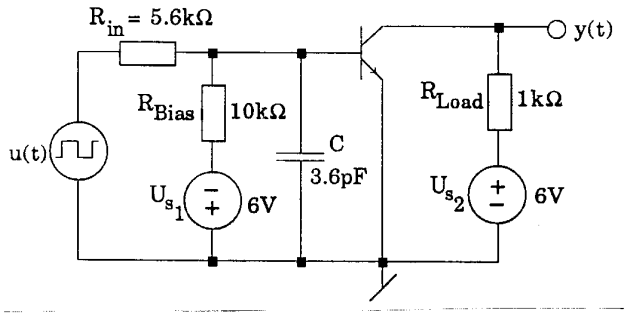

Figure H6.7. Schematic of a logic inverter model

For the transistor model, we want to use the default values except for the following parameters:

$$BF = 190.0$$
$$BR = 0.1$$
$$TF = 50\ psec$$
$$TR = 1\ \mu sec$$
$$IS = 1.1\ fA$$
$$RB = 1109.9\ \Omega$$

However, before you can invoke the DYMOLA preprocessor, you will have to modify the model manually to some extent. Remove the external base–collector capacitance, and replace the base resistance by a regular resistor. Then, you will also have to modify the junction diode model. Since we know that we must solve the diode equation for u_d rather than for i_d, we must invert this non–linear equation manually. Simplify the equation until it has an analytical inverse, and invert it. You may also want to limit the

diode current i_d to prevent the inverted diode equation from ever trying to compute the logarithm of a negative number. With these modifications, DYMOLA will be able to handle the transistor model.

I suggest that you print out the equations, but then immediately use the *eliminate parameters* command followed by the *eliminate equations* command to get rid of the many remaining defaulted parameters and trivial equations. Observe what happens to the equations on the way. Thereafter, *partition* your equations, and observe the *sorted equations* and the *solved equations*. Then generate a simulation model, and simulate the circuit during 150 *nsec* with a step size of 15 *psec* using either DESIRE [6.5] or SIMNON [6.2]. The pulsed input is supposed to be $u(t) = 0.0\ V$ except during the time from 20 *nsec* to 100 *nsec*, when the voltage is $u(t) = 7.0\ V$.

Now return to your original DYMOLA program, and add two more parameters:

$$RE = 13.3\ \Omega$$
$$RC = 750.0\ \Omega$$

Notice that you must return to your original DYMOLA program since both parameters had been optimized away with the *eliminate parameters* command. Repeat the same sequence of operations as before. You will observe that, this time, an algebraic loop occurs. After partitioning the equations, apply the various elimination algorithms, and observe what happens to the algebraic loop. After the equations have been totally optimized, analyze the algebraic loop using CTRL–C (MATLAB). You will notice that all loop variables appear linearly in all loop equations, i.e., this is a linear algebraic loop. Concatenate all loop variables into a column vector, **x**, and concatenate all previously computed variables into another column vector, **u**. Using this notation, the algebraic loop can be written in a matrix form as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{B} \cdot \mathbf{u}$$

A and **B** are constant matrices that depend only on parameter values. Use CTRL–C (or MATLAB) to solve these equations numerically for the loop vector **x**:

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{B} \cdot \mathbf{u}$$

Replace the loop equations in your DESIRE [6.5] or SIMNON [6.2] program manually by the set of solved equations obtained from CTRL–C (MATLAB), and repeat the simulation as before. It is foreseen to eventually automate this procedure in DYMOLA by providing an *eliminate loops* command.

Projects

[P6.1] Transistor Models

From the HSPICE manual [6.8] (which, to my knowledge, is currently the most extensive and most explicit SPICE manual available), develop DYMOLA models for the MOSFET and JFET devices similar to the BJT (NPNlat) model presented in this chapter.

[R6.2] Linear Algebraic Loops

Design and implement an algorithm that will automatically solve all sets of linearly coupled algebraic equations using the mechanism advocated in hw(H6.7). Test this algorithm by means of transistor models that contain emitter and collector resistance values.

Research

[R6.1] Junction Capacitance Model

Paul Van Halen [6.10] proposed a new junction capacitance model that avoids problems with a singularity that was inherent in the original SPICE junction capacitance model. As we have seen, this model cannot directly be applied to a DYMOLA circuit simulator since the charge equation must be solved for the voltage v rather than for the charge Q_c. Unfortunately, neither the original SPICE junction capacitance charge equation nor the modified junction capacitance charge equation has an analytical inverse (except in the simplified case that was treated in hw(H6.7). Use curve fitting techniques to come up with yet another junction capacitance charge equation that computes the voltage across the junction capacitance v as a function of the charge Q_c stored in the junction capacitance.

Also, come up with a better mechanism to represent the distributed base–collector junction capacitance than that of an internal and an external capacitance, a mechanism that avoids the structural singularity inherent in the above separation.

The goal of this research is to minimize the number of algebraic loops and structural singularities in a transistor model by at least eliminating those that are currently inside one transistor model. However, as we have seen in Chapter 5, new algebraic loops and/or structural singularities may be introduced in the course of coupling different submodels together.