Semiconductor Modeling with Bondgraphs

by

Michael Schweisguth

A Thesis Submitted to the Faculty of the

Department of Electrical and Computer Engineering

In Partial Fulfillment of the Requirements
For the Degree of

Masters of Science

In the Graduate College

The University of Arizona

1 9 9 7

Statement by Author

Approval by Thesis Director

This thesis has been approved on the date shown below:

______________________________	______________________________
Dr. F.E. Cellier Professor of Electrical and Computer Engineering	Date

Acknowledgments

The most important acknowledgment that I thought I should make is to the Cannondale bicycle company who makes such wonderful bicycles. I probably enjoyed more hours of pleasure with my bicycle, the Catalina mountains, and the Tucson sun-shine than a man deserves.

Of all the things that I have encountered during my tenure at the University of Arizona, I remember Kant's *Grounding* with a perpetual fondness. I particularly liked Immanuel Kant's observation that happiness is not a product of the rational mind, but instead a product of the imaginative mind. For me, this concept provides the unification between art and science; Art encompasses our dreams, passions, and instincts, while science provides us with the methods to realize, grasp, and appreciate *our* art. Through the translation I read by James W. Ellington, I came to appreciate the omnipresent subtlety of the *yin* and the *yang* which is seemingly so present in man's perception of reality. I thank Mr. Sullivan, a homeless *hop head*, for helping me appreciate these works. This man taught me so much for only one *sour dough Jumbo Jack*, with cheese.

Besides Kant's *Grounding*, I have also come to cherish the *Phantom of the Opera*, which I saw in NYC. While I don't think that I've experienced pure science, I believe that this opportunity let me experience pure art.

Of course, this acknowledgment would not be complete if I did not mention my friends: Yakov, Daniel, Robert, Ed, Tom, ChengLu, Klaus, Nanda, Jeff, Doug, Len, Asher, and Joe. Thanks for being there for me. I would also like to acknowledge those folks who make the U of A possible. These souls include the custodial staff, the administrative staff, the faculty, the professional staff, and the tax payers. I would imagine that quite a few of these people may never have the luxury to take classes at the U of A, and so, I feel a deep obligation to thank them for making this sacrifice for me. I hope that they don't find it demeaning that I compare them to the plankton in the ocean which produce the oxygen which the bigger animals breathe; And hence, they are the fundamental building blocks of our society's future, not I.

While it's not nice to point fingers, I must point out that I really enjoyed my computer science classes with Greg Andrews, Sampath Cannon, Richard Schlichting, and Larry Peterson. I must also point my fingers at Hal Tharp whose vector space lectures still seem to thrill my subconscious mind. To Chris Kostoff and Tom Milster, I flash a nice big smile. Chris was my guitar instructor; He taught me a lot about everything. Tom did too, and I appreciated the opportunity to work in his *high performance optical systems* laboratory.

And finally, I want to thank my advisor, Francois Cellier, for his patience, and for just being there.

Dedication

Dedicated to My Grandmother, the Sonoran Desert, My Cannondale Bicycle, My Guitar, Ghandi, and Mother Theresa.

Table of Contents

TABLE OF CONTENTS—*Continued*

List of Figures

List of Figures—*Continued*

Chapter 1
Introduction

Jim Williams is an editor of a rather interesting series of books which are filled with a bunch of marvelous stories about the history of analog circuit design. Each story is fable like– giving the reader a good dose of wisdom. One of my favorite stories revolves around the Wein Bridge Oscillator Circuits which Hewlett and Packard developed and then used to build HP's first line of waveform synthesizers. These stories, from this author's perspective, get the reader all pumped up and fantasizing about being an Analog Circuit Designer. That is, until the reader finds the list of questions which the industry asks its new-hires.

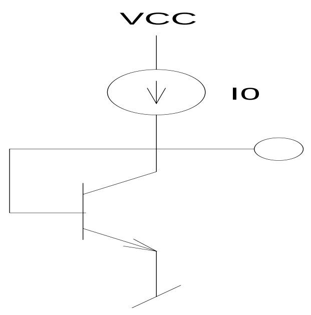

Figure 1.1. Transistor Circuit for New Hire

One of these questions concerns the operation of a transistor. In particular, the question states: "What happens to V_{out} as the temperature of the transistor in figure 1.1 increases?" The interviewer expects that a good analog engineer would give the answer that the transistor is configured as a diode, and that the voltage across the diode would decrease two millivolts per degree C. Hence, because the current remains constant, and the voltage decreases, the transistor would dissipate less power. While this answer apparently satisfies an interviewer, it does not ask the followup question: "what part of the circuit must now dissipate more power, since the voltage source and current source are constant?" The classical response to this question would be that the sources are not really ideal, and therefore, they contain a small amount of resistance; This small resistance must therefore dissipate this power. However, if the thermodynamic behavior of the circuit is also considered, then electrical power can also be transformed into heat.

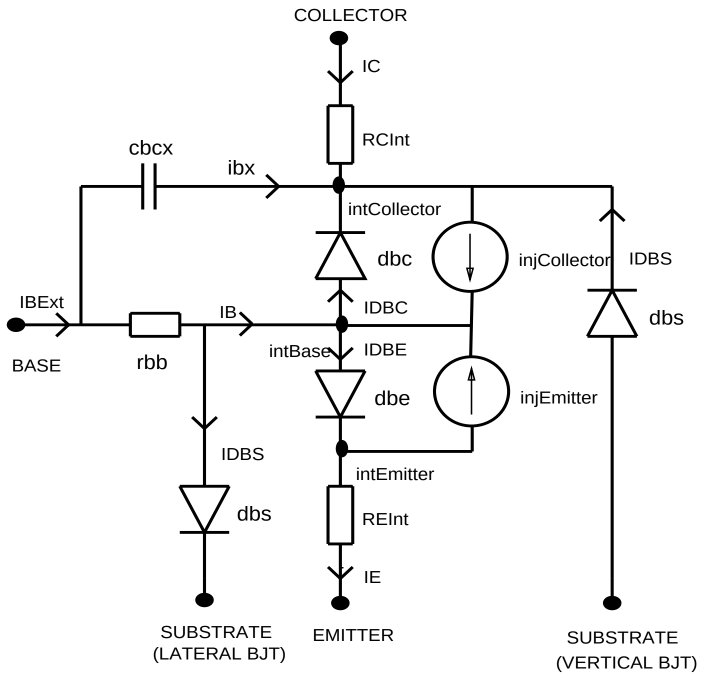

FIGURE 1.2. Transistor Circuit Model

The work contained in this thesis is the conversion of a standard "electronic circuit based" transistor model into a "bond graph based" model. This bond graph model not only allows the modeler to determine the model's voltage and current trajectories, but also its power flow trajectories. While power flow can be derived from the voltage and current flows in an electrical circuit, a bond graph model uses power flow as a first principle. This first principle states that power flow is simply the rate at which energy flows between localized points in space which have different energy levels; And, energy in a closed system always remains constant. Because of these principles, a transformer can be used to allow power to flow between different types of systems such as electrical and thermal. Given these properties of a "bond graph based" transistor model, all of the questions which were raised in the preceding paragraph can be answered.

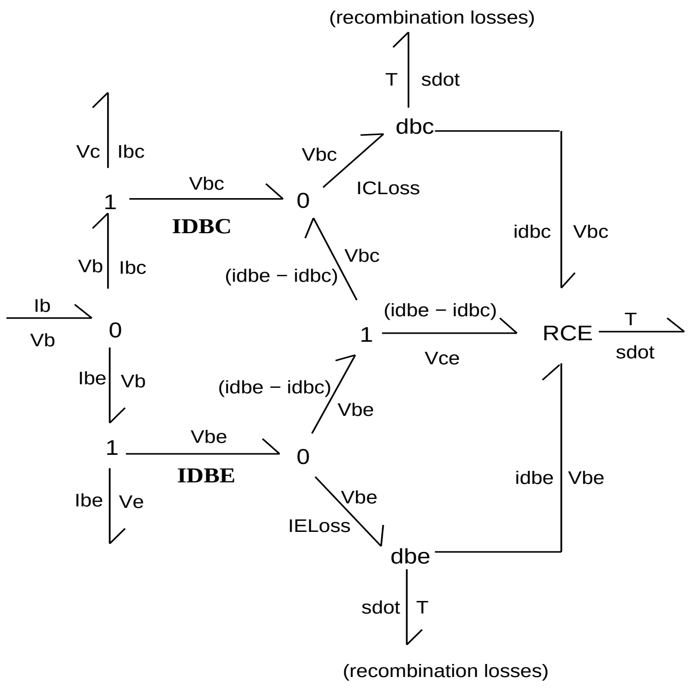

FIGURE 1.3. Transistor Bond Graph Model

The major references for this work can be found in [7], [1], and [4]. These sources develop the Gummel-Poon Transistor model which is described in [9] . The result of these works is an "electronic circuit based" model of a transistor which is nearly the same model as the one shown in figure 1.2. The Gummel-Poon model is used because it is a very popular model and it is used to predict the trajectories of real transistor operation in circuit simulation packages such as: *SPICE*, *PSpice*, and *BBSpice*.

While these circuit simulation programs allow the modeler to specify the initial temperature of the environment, they do not adjust the temperature of the environment during the simulation. That is, these simulation programs do not modify the simulation temperature to reflect the fact that the circuit's resistors are heating up and adding heat to the environment. The bond graph model which is developed in this thesis, on the other hand, lets the environment heat up. Section 5.5 shows the feedback effect of the ”heated up environment” on the opamp circuit’s output.

The bond graph models are developed using the Dymola modeling environment. This environment is the subject of chapter 2. Chapter number 3 provides a background of bond graph models. The process of translating the BJT circuit model in figure 1.2 to the bond graph model shown in figure 1.3 is discussed in chapter 4. The final chapter provides a demonstration of the bjt bond graph through the simulation of a simple inverter circuit, a simple buffer circuit, and a more complex opamp circuit.

Chapter 2
Modeling in Dymola

2.1 Introduction

Computer simulation is an important step in verifying an engineering design. To this end, all areas of engineering have computer based simulation programs which are designed to solve domain specific problems in an efficient way.

The design tool that was chosen to create a bond graph model of a BJT, with a thermal interface, was Dymola. The Dymola environment consists of: the Dymola Compiler which creates a simulation executable from a model described in the Dymola modeling language; the *Dymo Draw* editor which allows the modeler to create and maintain Dymola models using graphical tools; and *Dymo View* which allows the modeler to view the results of his or her simulation. These three tools, together, are collectively referred to as the Dymola modeling environment.

2.2 The Equation Sorter

The Dymola language is not a procedural language. Instead, the Dymola language is a-causal. Hence, the modeler is relieved from having to directly assign causality to each variable. As an example, consider the equation for Ohm's Law:

$$V = I * R \tag{2.1}$$

Because the Dymola compiler treats this equation as an a-causal statement, the Dymola compiler can symbolically derive a solution for the variable V, I, or R. This means that the modeler does not have to generate and work with three different forms of the equation. Instead, the modeler can use a single form of a given fundamental relationship and let the Dymola compiler handle the algebraic manipulation if it is required. If Dymola's solution is non-linear, then a non-linear equation solver will be employed automatically.

The Dymola compiler also uses an equation sorter which means that the modeler does not have to collate the model's equations by hand. Instead, the Dymola compiler will read in any random permutation of the model's equations and then sort them into an ordered sequence of equations. This ordered sequence of equations is then used by the compiler to generate a list of causal statements that calculates a solution trajectory for each of the model's unknown variables.

Because the Dymola compiler treats each equation as being an algebraic statement, the modeler must be careful when using equations which have the following form:

$$sampletime = sampletime + dt \tag{2.2}$$

If such a statement was encountered by a C compiler, the C compiler would interpret it as:

$$sampletime(t+1) \quad = \quad sampletime(t) + dt \tag{2.3}$$

Intuitively, one might think that the Dymola compiler would interpret the proposed equation as:

$$dt = 0 \tag{2.4}$$

because the variable *sampletime* can be subtracted from each side of the equation. However, the Dymola compiler derives a different result:

$$\begin{aligned} sampletime &= sampletime + dt \qquad (2.5)\\ sampletime * (1-1) &= dt \\ sampletime &= \frac{dt}{0} \end{aligned}$$

The solution which the Dymola compiler obtains, for the variable *sampletime*, shows just how disastrous the proposed equation can be if it is used in a model. A simulation of this equation was performed and the simulation resulted in finding a value of 10^{300} for the variable *sampletime*. Hence, the underlying floating point representation had no problem of storing the dubious result predicted by Dymola's equation. While this may seem like a trivial example, it demonstrates just how difficult it may be to debug a large scale model which has many *automatically* derived formulas.

If the modeler wants the Dymola compiler to treat the variable *sampletime* like a C compiler does, then the following notation must be used:

$$new(sampletime) = sampletime + dt \tag{2.6}$$

By formulating the equation with this syntax, the Dymola compiler is informed that the equation needs to be treated in a special way. In particular, the compiler will modify equation 2.6 so that it has the form:

$$newsampletime = sampletime + dt \tag{2.7}$$

By using an auxiliary variable, the equation sorter is then able to sort this equation along with the model's other equations.

The variable *sampletime* is called a discrete state variable because it's new value is calculated at discrete time intervals. The simulation of this equation found that *sampltime* had a value of zero. This is because *sampletime* had an initial state of zero. In order for the proposed equation to get reevaluated, the modeler *must* use discrete events. The Dymola modeling language supports the concept of *discrete events* in a few different ways. The most explicit form is through the use of the **when** statement. The following Dymola code fragment shows one way of implementing the proposed equation in Dymola:

```
when sampletime < Time then
    new(sampletime) = sampletime + dt
endwhen
```

The body of the **when** statement is executed **when** *sampletime* becomes less than

Time. Hence, the value of *sampletime* is recalculated every *dt* seconds. The above **when** statement is called a *rescheduling event* because the event reschedules itself.

Another type of discrete event formulation found in Dymola is:

```
outputvoltage = if (voltage > 1.0) then 1.0 else voltage
```

In order to handle this statement properly, the Dymola compiler generates code which determines **when** the value for *voltage* becomes greater than 1.0 and **when** the value for *voltage* becomes less than or equal to 1.0. This type of *event* is unlike a *scheduled event* because it can occur at an arbitrary time. In order to detect this sort of *event*, the simulation engine must either use interpolation, or modify the step size which it is using. The extent to which the simulation engine employs these methods is based upon the desired accuracy of the simulation.

2.3 Models with State Variables

The Dymola compiler allows the modeler to use differential equations to describe his/her model's dynamic trajectories. Differential equations are used in many different types of models such as: electrical models, chemical models, and thermal models. In order to use differential equations, the modeler must use the following syntax:

$$i = C * der(v) \tag{2.8}$$

Equation 2.8 models the ideal electrical Capacitor. The syntax $der(v)$ implicitly declares v to be a state variable. When the Dymola compiler finds a state variable, it will try to calculate the trajectory of the state variable by applying integration. This is because numerical differentiation is inherently unstable [13]. Hence, the Dymola compiler always tries to solve for the variable v (equation 2.8) by integrating $\frac{i}{C}$.

A problem that can be encountered when a model contains state variables is that the model might have too many state variables. That is, some of the state variables are linearly dependent on the others. If this situation occurs, the Dymola compiler will issue an error message that the model has too many state variables. The Dymola compiler has a special command which allows the modeler to simulate a model with this problem. This problem was not encountered with the models considered in this thesis. Hence, the solution to this problem is not discussed here.

The integration algorithm which the Dymola simulation engine uses to perform the integration is not fixed. Therefore, the modeler has the freedom to choose which integration algorithm is actually used to do the integration. The integration algorithm library contains the most popular standard integration algorithms. The program's default integration algorithm was used.

The discrete events, discussed previously, have an effect on the way these continuous integration algorithms work. In particular, when a discrete event occurs, the Dymola simulation engine sees this situation as signaling a new set of initial conditions for the model being simulated. If this procedure is not followed, then the step size control algorithm must try to capture the discontinuity caused by the discrete event. Depending on the type of discontinuity introduced by the discrete event, the process of reducing the step size might not converge on a result that meets a specified accuracy requirement. Hence, the simulator would be forced to stop the simulation on these grounds.

2.4 Algebraic Loops

Figure 2.1 shows a simple electrical circuit which has an algebraic loop. That is, a system of equations (2.9) is needed to determine the values of the variables $I1$ and $I2$ if they are the unknown variables in the system.

$$\begin{bmatrix} V1 \\ -V2 \end{bmatrix} = \begin{bmatrix} (R1+R3) & -R3 \\ -R3 & (R2+R3) \end{bmatrix} \begin{bmatrix} I1 \\ I2 \end{bmatrix} \tag{2.9}$$

The Dymola compiler is fully capable of detecting algebraic loops and has two different ways that it can deal with them. If the system of equations is small, then the

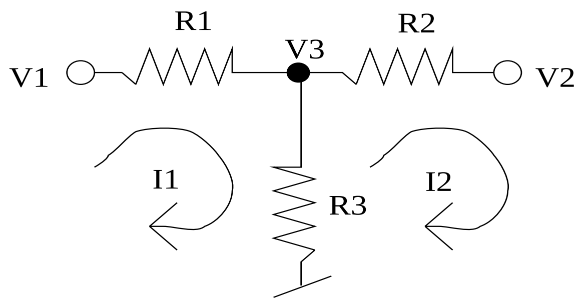

FIGURE 2.1. Circuit with Algebraic Loop

Dymola compiler can generate a symbolic solution which does not require the system matrix to be inverted. However, as the number of unknowns increase, the number of symbolic equations becomes very large. Hence, the Dymola compiler will need to build a system matrix and then invert it to calculate the necessary unknowns. This technique is very general, and it even allows the Dymola simulation engine to work with a system of non-linear equations.

2.5 The Dymola Object Model

The most basic building block in Dymola is the *model class* and it has the following syntax in a flat text file:

```
model class  ComponentName
end
```

The Dymola language allows the modeler to use inheritance when creating *model classes*. The following example shows the basic form of inheritance:

```
model class  BaseClass
    cut pin1(vin, current)
    cut pin2(vout, -current)
end

model class (BaseClass)  DerivedClass
end
```

In this example, **model class** *DerivedClass* inherits the **cuts** from the **model class** *BaseClass*. Every variable declared in the base class is publicly accessible to the derived class.

2.6 Object Communication

There are 3 different ways that Dymola objects can communicate. The first is through the concept of *cuts*, the second is through the concept of *terminals*, and the third is by using global variables.

The basic declaration for a **cut** in Dymola is:

```
model class ComponentName
    cut connector(A1,A2, ..An/ T1,T2, ..Tn)
end
```

Objects with connectors can be connected together with the syntax:

```
connect object1:connector at object2:connector
```

so they can communicate through their *cuts*.

The **cut** parameters labeled $A1$ through A_n are called *across* variables, while the **cut** parameters labeled $T1$ through T_n are called *through* variables. When the Dymola compiler encounters a **connect** statement, it generates the following equations using the cut's *across* variables:

$$\begin{aligned} object1.A1 &= object2.A1 = ... = objectN.A1 \\ object1.A2 &= object2.A2 = ... = objectN.A2 \\ ... &= ... \\ object1.AN &= object2.AN = .. = objectn.AN \end{aligned} \tag{2.10}$$

while it generates the following equations using the cut's *through* variables:

$$\begin{aligned} object1.T1 \;&+\; object2.T1 + ... + objectN.T1 = 0 \\ object1.T2 \;&+\; object2.T2 + ... + objectN.T2 = 0 \\ ... \;&+\; ... \\ object1.Tn \;&+\; object2.Tn + ... + objectN.Tn = 0 \end{aligned} \tag{2.11}$$

Figure 2.2 shows a very simple electrical circuit. The *across* variables for this circuit are the potentials $V1$ and $V2$ while the *through* variables are the currents iR_1, iR_2, iR_3, i_{in}, and i_{out}. If the resistors in figure 2.2 have the following Dymola interface:

```
model class Resistor
    cut pin1(vin/ i)
    cut pin2(vout/ -i)
end
```

and the following connection equations were used:

```
cut InputPin (v1, -iIn)
cut OutputPin (v2, iOut)

connect InputPin at R1:pin1
connect R1:pin1 at R2:pin1
connect R2:pin1 at R3:pin1

connect OutputPin at R1:pin2
connect R1:pin2 at R2:pin2
connect R2:pin2 at R3:pin2
```

then the Dymola compiler would generate the following *Across Equations*:

$$\begin{aligned} R1.vin = R2.vin = R3.vin = InputPin.V1 \\ R2.vout = R2.vout = R3.vout = OutputPin.V2 \end{aligned} \tag{2.12}$$

and the following *Through Equations*:

$$\begin{aligned} -InputPin.iIn + R1.i + R2.i + R3.i &= 0 \\ OutputPin.iOut - R1.i - R2.i - R3.i &= 0 \end{aligned} \tag{2.13}$$

The equations shown in 2.12 and 2.13 are produced automatically by the Dymola compiler because of the **connect** statements. Hence, equations 2.12 and 2.13 are called the model's connection equations.

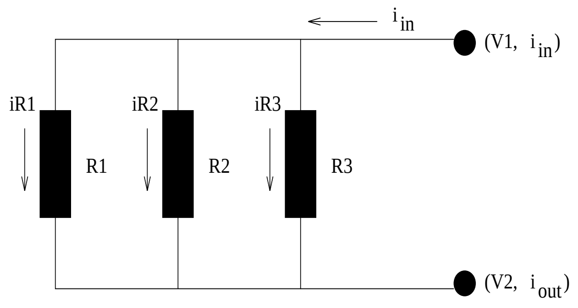

FIGURE 2.2. Simple Electric Circuit

The modeler may not want to generate all these equations. Hence, there is a special syntax which can be used to inform the Dymola compiler that it should not generate certain equations automatically. The following Dymola code fragment illustrates this concept.

```
model class component
    cut connector(v/.)
end
```

By using a period in place of a variable name, the Dymola compiler will have one less variable to solve for. It may be necessary to use the period when you declare a **cut** to make that **cut** compatible with another **cut**. In Dymola, **cuts** can only be connected together if they have the same number of *across* variables and *through* variables. Hence, in the example shown, the period keeps the **cut** specification compatible with other **cuts** that have exactly one *across* variable and one *through* variable. If the **cut** declaration has many *across* or *through* variables, then periods may be used as place holders.

Another form of communication between Dymola objects is through the use of terminals. The Dymola language offers three ways to specify a terminal:

```
terminal inout
input in
output out
```

The way a *terminal* is declared effects its causality. If a *terminal* is declared with the keyword **terminal**, then the Dymola compiler can calculate the value of the *terminal* variable inside of the object or outside of the object. If a *terminal* is declared with the keyword **output**, then the value of the *terminal* variable is calculated within the object. Finally, if a *terminal* is declared with the keyword **input**, then the value of the *terminal* is calculated outside of the object.

An object's terminals are accessed using *dot notation*. If the following Dymola code is used to model a resistor:

```
model class Resistor
    cut(vin,current)
    cut(vout, -current)
    output voltagedrop
    voltagedrop = vin - vout
end
```

then the *output* terminal would be accessed using:

```
Resistor.voltagedrop
```

The last method of communication between objects is through the use of global variables. If an object wishes to use a global variable, then it uses the syntax:

```
external variablename
```

If an object wishes to provide the value for a global variable, then it uses the following notation:

```
internal variablename
```

2.7 Instantiation of Dymola Objects

Each Dymola object can have a set of instantiation parameters. These parameters are used by the object to customize itself in some way. A typical parameter would be the specific resistance for a specific resistor. Parameters are declared using the following Dymola syntax:

```
model class Resistor
    parameter R = 1
end
```

When a parameter is declared, it can be given a default value. When an object is instantiated, with the keyword **submodel**, the modeler may choose to override the object's default parameters. The following Dymola code shows how this is done:

```
submodel Resistor Rload(R=1997)
```

The modeler may wish to customize existing objects using the object oriented concept of $HAS - A$. The following Dymola code shows an implementation of the $HAS - A$ relationship.

```
model class OneK
    cut pin1(vin/ current)
    cut pin2(vout/ -current)
    parameter R = 1000
    submodel Resistor R(R=R) {OneK HASA Resistor}
    R.vin = vin {make connections between classes}
    R.vout = vout
    R.current = current
end
```

In this example, the **model class** structure is used to provide a different default value for the **parameter** R. This design style is used extensively to create various flavors of NPN and PNP transistors which only differ in the instantiation parameters.

2.8 A Small Demonstration Model

This section presents a *small* Dymola model using some of the syntax that was presented in the previous sections.

The circuit that is constructed in this section is shown in figure 2.3. The circuit is a simple RLC circuit which consists of a frequency generator, two resistors, a capacitor, and an inductor. During the simulation, the frequency generator increases the frequency of its output. Hence, the voltage across the capacitor will identify the circuit's resonance frequency.

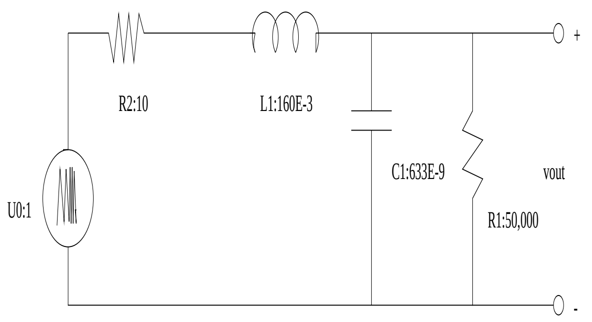

FIGURE 2.3. Simple RLC Circuit

The base class of the *Resistor*, *Capacitor*, *Inductor* is a class called *Component*. The idea is that the input to the component is *pin*1, while the output of the component is *pin*2.

```
model class Component
    cut pin1(voltageIn/current)
    cut pin2(voltageOut/-current)
    local voltagedrop { local means local variable }
    voltagedrop = (voltageIn - voltageOut)
end
```

The *Resistor* class is derived from the **model class** *Component*. The equation used for the ideal resistor model is Ohm's law. The *Capacitor* is also derived from the **model class** *Component*. The equation used for the ideal capacitor is $C*der(\mathrm{V}) = i$. Finally, the *Inductor* is also derived from the **model class** *Component*. The equation used for the ideal inductor is $L*der(\mathrm{i}) = \mathrm{V}$.

```
model class (Component) Resistor
    parameter R
    voltagedrop = R*current
end

model class (Component) Capacitor
    parameter C
    current = C*der(voltagedrop)
end

model class (Component) Inductor
    parameter L
    voltagedrop = L*der(current)
end
```

The last **model class** is called *FrequencySweep*. The output of this component is a sinusoid. The frequency of the sinusoid is increased every *dt* seconds. (see figure 2.4)

```
model class FrequencySweep
    parameter f0initial = 200
    parameter deltaf0 = 100
    parameter amplitude = 0.707
    parameter updateinterval = 0.1
    local nexttime = updateinterval
    local sinf0 = f0initial
    cut pin1 (voltageIn/.)
    cut pin2 (voltageOut/-current)
    voltageIn = 0
    voltageOut = amplitude*sin(2*3.1459*sinf0*Time)
    when (nexttime < Time) then
        new(sinf0) = sinf0 + deltaf0
    endwhen
end
```

After the **model classes** are declared, a **model** can be created. The Dymola compiler can only construct one **model**. The **model**, however, can use many **model classes**. The following Dymola **model** uses the *Resistor*, *Capacitor*, *Inductor*, and *FrequencySweep* **model classes** to build the desired high pass RCL filter shown in figure 2.3.

```
model RLCFilter
    submodel (FrequencySweep) SRC − >
```

```
        (f0initial=200, deltaf0=100, amplitude=1, updateinterval=0.1)
    submodel (L) L1(L=160E-3)
    submodel (R) R1(R=50000) R2(R=50)
    submodel (C) C1(C=633E-9)
    connect R2:pin2 at L1:pin1
    connect L1:pin2 at C1:pin1
    connect L1:pin2 at R1:pin1
    connect R2:pin1 at SRC:pin1
    connect SRC:pin2 at C1:pin2
    connect C1:pin2 at R1:pin2
end
```

2.9 Simulating the Model

The Dymola compiler provides the user with a command line interface that allows the user to control the Dymola compiler. The following is the list of commands that were used to simulate the RLCFilter model.

```
enter model rclfilter.dym
partition
output model
compile
set variable value RLCFilter::C1.e=0.0 {initial state value}
set variable value RLCFIlter::L1.e=0.0 {initial state value}
experiment StopTime = 1.0
output experiment
simulate
```

The command **enter model** reads in the model. The model is coded in the Dymola modeling language. The command **partition** sorts the model's equations and assigns causality to each of the model's variables. The command **output model** outputs a model. The default output language is "C." The **compile** command causes Dymola to invoke the gcc compiler to compile the model and link it with its simulation libraries. The command **output experiment** creates an experiment file. This file contains information such as: how many communication points are desired, what integration algorithm should be used, the initial values of the systems variables, etc... The final command, **simulate**, causes the simulation's executable to be executed. The products of these commands are:

- dsmodel.c - This is the C code which implements the model coded in the Dymola modeling language. This file is produced when the **output model** command is issued.
- dymosim - This is the simulation executable produced by compiling dsmodel.c and then linking dsmodel.o with Dymola's simulation library. Dymosim is created by the **compile** command.
- dsres.mat - This file contains the results of the simulation. This file is created by the running the dymosim executable with the **simulate** command.
- dsin.txt - This file contains things like the parameter values for the model's objects, the desired simulation time, the number of data points that should be saved, the initial values of the model's state variables, etc. The file is created with the **output experiment** command. It is used by the *dymosim* executable. Because dsin.txt is a text file, the modeler may modify it with a text editor.

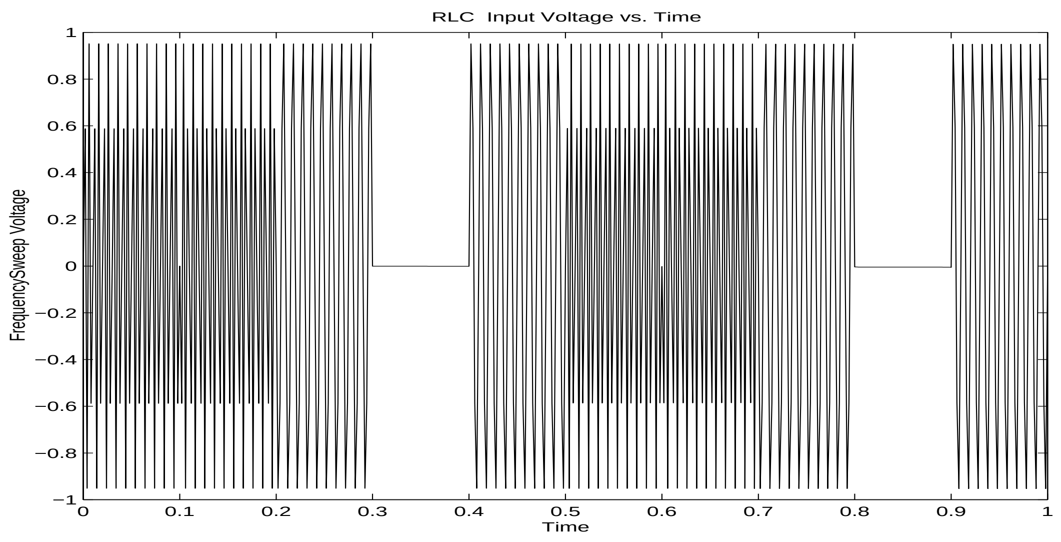

FIGURE 2.4. Plot of RLC Input Voltage

After the simulation was finished, MATLAB was used to analyze the trajectory of the capacitor voltage as a function of frequency (figure 2.6).

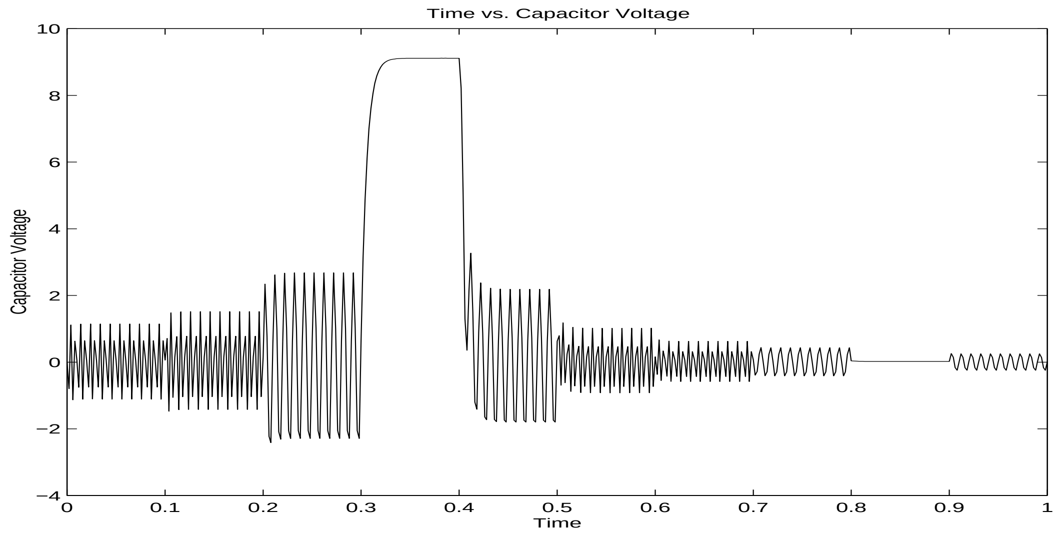

FIGURE 2.5. Plot of Capacitor Voltage vs. Time

2.10 Dymo Draw

Although the above **model** and **model classes** can be entered into a text editor by hand, the Dymola modeling environment offers a graphical editor called *Dymo Draw*. *Dymo Draw* allows the modeler to create and maintain large scale models with a GUI. Each **model class** can have a custom icon associated with it. This makes it easier for other modelers to reuse and maintain various parts of a simulation code. The following Dymola code shows an RCfilter model with graphical layout information:

```
model RCFilter {* (-100, -100) (100, 100)}
    {* window 0.33 0.22 0.6 0.6 }
    submodel (FrequencySweep) FS {* at (-120, -20) (-48, 62) rotation=-90 }
    submodel (Resistor) Resistor {* at (-54, 90) (18, 42)} (R=1000)
    submodel (Capacitor) C {* at (6, -24) (90, 58) rotation=-90 }
    connect FS:pin2 at C:pin2 {* via (-81, -19.7) (52, -19.8)}
    connect Resistor:pin2 at C:pin1 {* via (14, 68) (48, 68) (48, 54)}
    connect FS:pin1 at Resistor:pin1 {* via (-84, 60) (-84, 68) (-50, 68)}
    {* text "V" (82, 36) (122, 8)}
    {* line (46, -20) (92, -20)}
    {* ellipse (92, -14) (104, -26)}
    {* ellipse (92, 74) (104, 62)}
    {* line (46, 68) (92, 68)}
end
```

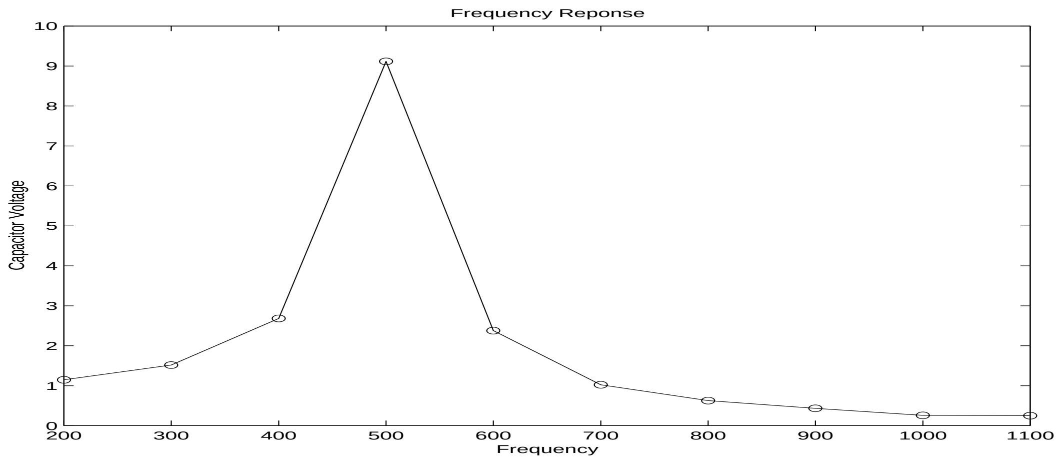

FIGURE 2.6. Plot of Capacitor Voltage vs. Frequency

While the *Dymo Draw* environment was used to design the BJT model, the graphical layout information will be striped from the listed source code in this thesis.

2.11 Debugging

The Dymola compiler offers a few ways to help debug the model. The hardest way is to look at the C code. The more useful debugging aid is to ask the Dymola compiler to print a list of the model's equations. The following options are available:

- output equations
- output sorted equations
- output solved equations

The process of debugging a Dymola model includes both run time errors and compile time errors. A run time error, such as taking the square root of a negative number, is handled while the simulation is running.

Compile time errors can be hard to find. In particular, a model might not specify enough equations to uniquely determine a trajectory for each of the model's variables. If the Dymola compiler encounters this problem, it is capable of telling the modeler which variables it could not derive an equation for. However, the Dymola compiler is not able to highlight the lines of source code that contain the missing equations that it needs. The process of correcting this problem begins with the equation list generated by the Dymola compiler. By reading this list of equations, the modeler is likely to determine that an equation is missing from a **model class** or that a **connect** statement is needed to generate missing connection equations.

The opposite problem may also occur. In particular, a model may provide too many solutions for a given unknown variable. When the Dymola compiler detects this, it will issue an error message that the system of equations is *over determined.*

2.12 Additional Information on Dymola

Additional information on the Dymola modeling environment can be found in the book Continuous System Modeling [4] or the Dymola Homepage [8].

Chapter 3
Bond graphs

3.1 Introduction

Bond graphs were introduced by H.M. Paynter in 1960 as a more general way to graphically represent a system [3, 4]. As an alternative to signal flow graphs and block diagrams, the bond graph allows the modeler to simultaneously show the topology and the computational structure of his or her model. The notation is general enough so that it can be used to represent many different kinds of models. These include chemical, electrical, thermal, mechanical, and others. [4, 13]

The two primary variables used in bond graph modeling are $effort$ and $flow$. Voltage, temperature, force, torque, and pressure are examples of effort variables, while current, entropy flow, velocity, angular velocity, and fluid flow are examples of flow variables [2]. The primary sets of $effort$ and $flow$ variables which are discussed in this thesis are voltage and current, which are used to model electrical systems, and temperature and entropy flow which are used to model thermal systems.

The topics found in this chapter include basic bond graph theory and exactly how bond graph models can be practically implemented on top of the Dymola modeling language. Additionally, the topic of creating bond graph models which include both electrical and thermodynamic equations will be discussed.

3.2 Bond Graph Basics

A bond graph is made up of, primarily, four different entities: bonds, 0-$junctions$, 1-$junctions$, components, and transformers.

Figure 3.1 shows what a bond looks like. The barb shows the direction of power flow. Each bond is associated with two variables. Mainly, one $effort$ variable (e) and

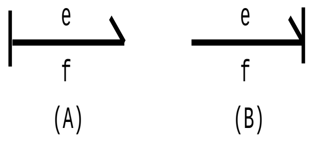

FIGURE 3.1. Bonds with Causality Strokes

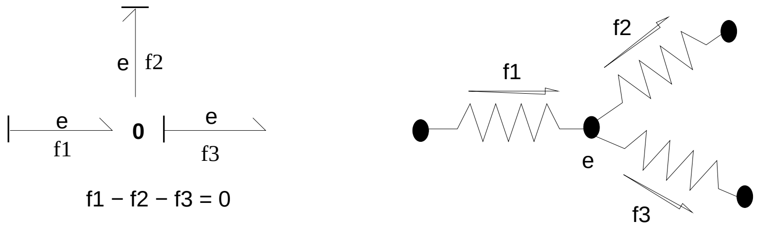

FIGURE 3.2. Graphical Representation of a Zero Junction

one $flow$ variable (f). In figure 3.1, the vertical bars show where the $flow$ variable is calculated. In figure 3.1.(A), the $flow$ is calculated at the input to the bond while in figure 3.1.(B) the $flow$ is calculated at the output of the bond. These vertical bars are called causality strokes. By adding a causality stroke to a bond, the bond is able to communicate the computational structure of the model.

In figure 3.2, a 0-*junction* is shown and in figure 3.3 a 1-*junction* is shown. Each of these junctions have three bonds attached to them. The 0-*junction* sums the flows that are associated with the attached bonds:

$$f_1 + f_2 + f_3 + f_4 + f_5 + ... + f_n = 0 \tag{3.1}$$

From inspection, equation 3.1 allows for the solution of exactly one $flow$ variable. Hence, the values of the other flow variables must be known. Therefore, only one causality stroke is shown at the 0-*junction* in figure 3.2.

Unlike the 0-*junction*, the 1-*junction* (figure 3.3) sums the efforts:

$$e_1 + e_2 + e_3 + e_4 + e_5 + ... + e_n = 0 \tag{3.2}$$

Again, by inspection, it can be determined that equation 3.2 allows for the solution

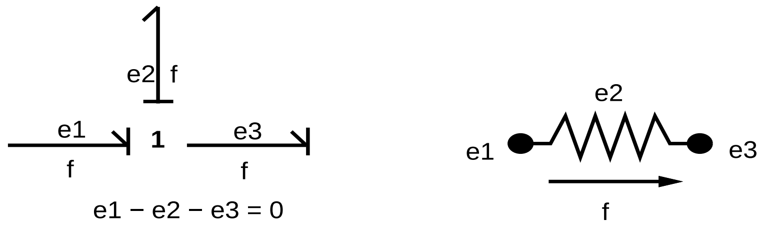

FIGURE 3.3. Graphical Representation of a One Junction

of only one effort variable. Hence, the values of the other effort variables must be calculated elsewhere.

Because *effort* (e) is constant at a 0-*junction* and *flow* (f) is constant at a 1-*junction*, it can be shown that the presented equations, for both the 0 and the 1 *junction*, conserve power:

$$\begin{aligned} e * (f_1 + f_2 + f_3 + f_4 + f_5 + ... + f_n) &= \sum_{i=0}^{n} e * f_i = \sum_{i=0}^{n} p_i = 0 \\ f * (e_1 + e_2 + e_3 + e_4 + e_5 + ... + e_n &= \sum_{i=0}^{n} f * e_i = \sum_{i=0}^{n} p_i = 0 \end{aligned} \tag{3.3}$$

In equation 3.3, p_i is simply a partial power.

3.3 Bond Graph Components

A bond graph component is said to provide the model's constitutive equations. There are several main classes of bond graph components. They include, but are not limited to: sources, resistors, and energy storage components.

There are two types of sources that are used in bond graph modeling. The first is called an *effort* source. The *effort* source is modeled as:

```
model class SE
    cut (e/.)
    e = e(t)
end
```

where $e(t)$ is an arbitrary function of time. The second type of source that is used is a *flow* source. The *flow* source is modeled as:

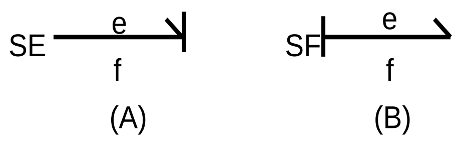

FIGURE 3.4. Causality of Flow and Effort Sources

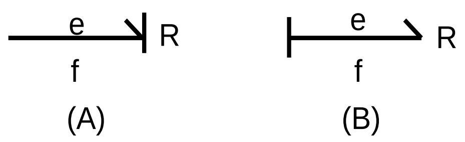

FIGURE 3.5. Causality of Resistors

```
model class SF
    cut (./-f)
    f = f(t)
end
```

To an electrical engineer, a well known effort source (SE) is a voltage source (*battery*), while a well known flow source (SF) is a direct current source.

A source only offers a single equation for either $effort$ or $flow$. Hence, there is no choice during causality assignment. The causality strokes for the $effort$ source and the $flow$ source are shown in figure 3.4.

The next type of bond graph component is called the resistor. Resistors do not store energy. Instead, they only dissipate energy. The sample Dymola code, below, shows a typical linear resistor.

```
model class R
    cut (e/f)
    parameter R
    e = R*f
end
```

The causality of a resistor is flexible. That is, algebraic manipulation may be applied to solve for either e or f. Hence, the causality strokes for a resistor are not fixed. (see figure 3.5)

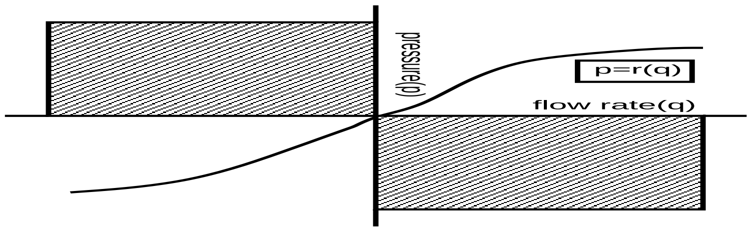

FIGURE 3.6. Non-Linear/Arbitrary Resistor

An important property of a resistor is that the algebraic signs of $effort$ (e) and $flow$ (f) are identical. That is, the product of e and f is always positive. This is because resistors always dissipate energy. Figure 3.6 shows an arbitrary non-linear resistor. In [4], a resistor is defined as a device that operates in the first and third quadrants. The plot in figure 3.6, for example, has this property.

The final type of bond graph component is the energy storage element. The two sub-components found in this class are the capacitor and the inductor. These devices are modeled as follows:

```
model class C
    cut (e/f)
    parameter C
    C*der(e) = f
end

model class L
    cut (e/f)
    parameter L
    L*der(f) = e
end
```

The capacitor and inductor are said to have derivate causality. That is, the solution trajectories for $effort$ and $flow$ are found through integration. The causality strokes for a capacitor and an inductor are shown in figure 3.7.

3.4 The Diamond Theorem

There are several methods to convert circuit diagrams to bond graphs. Two examples are shown in figures 3.2 and 3.3. The first example shows how Kirchoff's law (KCL)

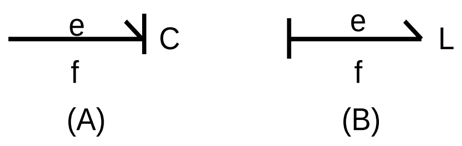

FIGURE 3.7. Derivative Causality of Capacitors and Inductors

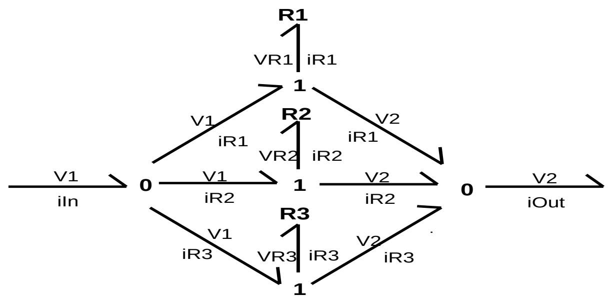

FIGURE 3.8. Parallel Resistors Bond Graph

is implemented with a 0-*junction*. The second example shows how a voltage drop across a resistor is modeled with a 1-*junction*.

As an additional example, the circuit shown in figure 2.2 is shown as a bond graph in figure 3.8. By applying the technique shown in figure 3.3, it can be determined that three one junctions are needed. Additionally, by applying the technique shown in figure 3.2, it can be determined that two 0-*junctions* are needed (two KCL equations). This bond graph representation is quite verbose and it can be simplified. Figure 3.9 shows the simplified bond graph.

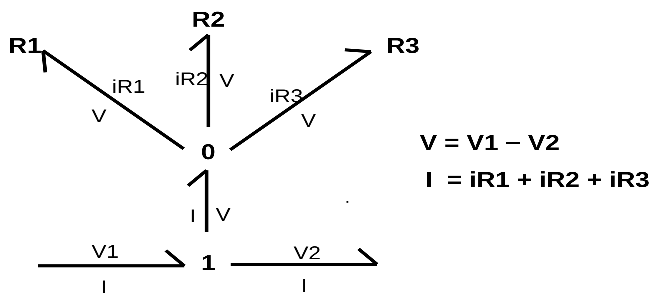

FIGURE 3.9. Parallel Resistors Bond Graph Using the Diamond Theorem

The transformation process of going from a bond graph that looks like the one in figure 3.8 to one that looks like the bond graph shown in 3.9 is formalized by the *Diamond Theorem*. This theorem basically recognizes that the voltage drop across each resistor is the same, so only one 1-*junction* is needed. The second part of the theorem is based on equation 3.3. Mainly:

$$e * (f_1 + f_2 + f_3) = 0 \tag{3.4}$$

3.5 The Simplectic Gyrator

Before Dymola had a graphical modeling environment (*Dymo Draw*), Cellier used the concept of the *simpletic gyrator* to implement bonds in Dymola. The *simpletic gyrator* has a trivial implementation and it is shown below:

```
model class Bond
    cut Bin(e/f)
    cut Bout(f/-e)
    { This class swaps the roles of e and f so that e can become a through variable
    and f can become an across variable. By using two bonds in series, the second bond
    undoes the swapping introduced by the first bond. Hence, e and f are again across
    and through variables at the output of the second bond. }
end
```

The bond graph component also has a simple implementation:

```
model class BGComponent
    cut B(e/f)
    { Components are implicitly zero junctions }
end
```

To implement a new bond graph component, a new component only needs to inherit the bond graph interface provided by the **model class** *BGComponent*. The following Dymola code illustrates this component design technique by showing how an ideal resistor would be declared:

```
model class (BGComponent) R
    parameter R
    e = R*f
end
```

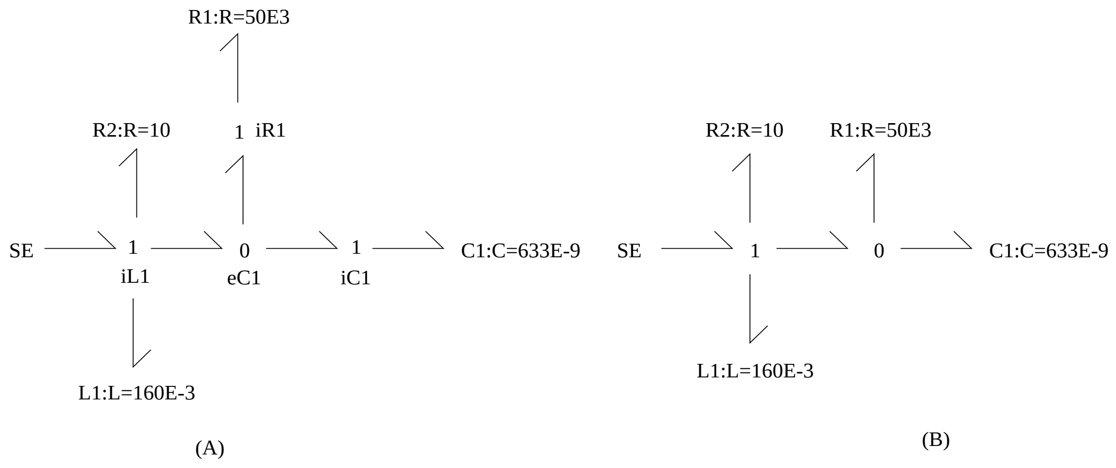

FIGURE 3.10. RLC Bond Graph Circuit

Finally, the non-graphical version of Dymola offered the modeler the ability to declare **nodes**. Nodes could be used like **cuts** and they acted primarily as private connection points within a **model class** or **model**. In Cellier's work, Cellier used nodes to implement 0 and 1 *junctions*. Because the *Dymo Draw* modeling environment does not include support for **nodes**, the following two **model classes** were designed to implement 0 and 1 *junctions*:

```
model class Zero
    cut B(./f)
    f = 0
end

model class One
    cut B(./e)
    e = 0
end
```

The model classes *Zero* and *One* are identical. They are duplicated to make debugging easier. In particular, *One* junctions are used when you want an equation which is homogeneous in e and *Zero* junctions are used when you want an equation which is homogeneous in f.

With these **model class** declarations, it is possible to implement the simple RLC circuit shown in 2.3. This circuit is modeled and shown using bond graph notation in figure 3.10.(A). It should be noted that the bond graph version of this circuit contains two additional 1-*junctions*. These 1-*junctions* are labeled $iC1$ and $iR1$. When the *simpletic gyrator* connects a component (an implicit 0-*junction*) to a 1-*junction*, it reverses the roles of the *across* (e) and *through* (f) variables. Because of this swapping, the Dymola compiler quietly generates the proper connection equations which will sum the *efforts* at the 1-*junction* instead of the *flows*. Therefore, a *simpletic gyrator* should not be used to connect a component to a 0-*junction*. Instead, a component should be attached directly to an adjacent 0-*junction* so that any swapping is avoided. This is because the Dymola compiler should generate connection equations which sum the *flows* at 0-*junctions*. While the *Dymo Draw* modeling environment certainly supports making a direct connection between objects, the resulting bond graph model would not have schematically pure bond graph notation. Thus, the technique of placing additional 1-*junctions* between the pairs of adjacent 0-*junctions* can be used to create a bond graph model which has a canonical form that conforms to the stipulation that 0 and 1 junctions have to alternate. This technique should also be applied to adjacent 1-*junctions*. In particular, a 0-*junction* should be inserted so that the junction types alternate properly. The Dymola source code, below, shows an implementation for all of the circuit parts which are needed to implement the *RLC* circuit.

```
model class (BGComponent) R
    parameter R=1.0
    e=R*f
end

model class (BGComponent) L
    parameter L=0.001
    L*der(f) = e
end

model class (BGComponent) C
    parameter C=0.001
    fracdedt = fracfC
end
```

```
model class SE
    parameter f0initial = 200
    parameter deltaf0 = 100
    parameter amplitude = 0.707
    parameter updateinterval = 0.1
    local nexttime = updateinterval
    local sinf0 = f0initial
    cut B (e/.)
    e = amplitude*sin(2*3.14159*sinf0*Time)
    when (nexttime < Time) then
        new(nexttime) = nexttime + updateinterval
        new(sinf0) = sinf0 + deltaf0
    endwhen
end

model RLC
    submodel (SE) SE − >
        (f0initial=200, deltaf0=100, amplitude=1, updateinterval=0.1)
    submodel (L) L1 (L=160E-3)
    submodel (Bond) b1 b2 b3 b4 b5 b6 b7 b8
    submodel (R) R1(R=50000) R2(R=50)
    submodel (C) C1 (C=633E-9)
    submodel (Zero) uC
    submodel (One) iC iR iL
    connect SE:B at b1:Bin
    connect iL:B at b1:Bout
    connect b3:Bin at iL:B
    connect L1:B at b3:Bout
    connect R2:B at b4:Bout
    connect b4:Bin at iL:B
    connect iL:B at b2:Bin
    connect uC:B at b6:Bin
    connect b5:Bout at C1:B
    connect iR:B at b8:Bin
    connect b8:Bout at R1:B
    connect b7:Bout at iR:B
    connect iC:B at b5:Bin
    connect b6:Bout at iC:B
    connect b2:Bout at uC:B
    connect uC:B at b7:Bin
end
```

3.6 A Smarter Simplectic Gyrator

There are several shortcomings to the technique described in section 3.5. The biggest inadequacy is that the component does not encapsulate the fact that it is a 0-*junction*.

Hence, the modeler is forced to remember this. If the modeler does not alternate the 0 and 1 junctions correctly, then the modeler has created a model that will compile ok, but it may not simulate properly. Hence, the design tool did not help the modeler create a correct model. From experience, the modeler will probably have to read the generated equation list to find out why the simulation produced incorrect results. An attempt was made to resolve these problems by redesigning the *Bond*, *BGComponent*, *Zero*, and *One* **model classes**. The new implementations for these **model classes** are shown below.

```
model class Bond
    cut Bin(ein, jtypein /fin)
    cut Bout(eout, jtypeout/-fout)
    local jnotsame
    jnotsame = (jtypein - jtypeout)**2
    eout = fin*jnotsame + ein*(1 - jnotsame)
    fout = ein*jnotsame + fin*(1 - jnotsame)
end

model class BGComponent
    cut B(e, jtype/ f)
    jtype = 0 { jtype = 0 for zero junction }
end

model class ZeroJunction
    cut B(., jtype/ f)
    jtype = 0 { jtype = 0 for zero junction }
    f = 0
end

model class OneJunction
    cut B(., jtype/ e)
    jtype = 1 { jtype = 1 for one junction }
    e = 0
end
```

The redesigned *bond* **model class** had this new feature: the bond was designed to be aware of the junction types that it was connecting together. This *awareness* is accomplished with the following equation in the *bond*:

$$jchange = (jtypein - jtypeout) * *2 \tag{3.5}$$

Because the values for $jtypein$ and $jtypeout$ are restricted to having a boolean value of 0 or 1, the variable $jchange$ can only have the following values:

$jtypein$	$jtypeout$	$jchange$
0	0	0
0	1	1
1	0	1
1	1	0

Hence, it is easy to see that the variable $jchange$ is also a boolean value which is 1 if the *bond* connects together different junction types or 0 if the *bond* connects together similar junction types. Therefore, the equation:

$$\begin{bmatrix} eout \\ fout \end{bmatrix} = \begin{bmatrix} (1 - jnotsame) & jnotsame \\ jnotsame & (1 - jnotsame) \end{bmatrix} \begin{bmatrix} ein \\ fin \end{bmatrix} \tag{3.6}$$

should reduce to either:

$$\begin{aligned} eout &= ein \\ fout &= fin \end{aligned}$$

if $jchange = 0$ or,

$$\begin{aligned} eout &= fin \\ fout &= ein \end{aligned}$$

if $jchange = 1$.

Unfortunately, this implementation does not work very well. The RLC circuit that was constructed with the redesigned *bond*, *BGComponent*, *Zero*, and *One* **model classes** is shown in figure 3.10.(B). Notice that this bond graph model does not have the property that 0 and 1 junctions alternate. If the Dymola compiler needed to generate a few additional run-time equations to support this new implementation, then an engineering trade off could be made between ease of graphical design versus a small run time penalty. Instead, this turned out not to be the case. In fact, the

Dymola compiler needed to generate a matrix solution to solve an algebraic loop which consisted of 14 unknown variables. Hence, the second technique of modeling bond graph models turned out to be a lot more expensive than the first technique (section 3.5) which only produced a series of 15 equations. According to Cellier, this occurs because the constants are combined after the causality is determined.

3.7 A Replacement for the Simplectic Gyrator

The last attempt at building a bond graph modeling toolbox for Dymola did not use the *simpletic gyrator* as a starting point. Instead, a different approach was used.

The *simpletic gyrator* used both *across* and *through* variables. This let Dymola generate the proper connection equations at each junction. In this last attempt, no *through* variables are used. Instead, the redesigned **model classes** implement the summations needed explicitly. The new *Bond*, *BGComponent*, *One*, and *Zero* classes are shown below:

```
model class Bond
    cut Bin(e,f, signIn)
    cut Bout(e,f, signOut)
    signIn = -1.0
    signOut = 1.0
end

model class BGComponent
    cut B(e,f,signf)
end

model class One4Junctions
    cut T1(e1, f, signe1)
    cut T2(e2, f, signe2)
    cut T3(e3, f, signe3)
    cut T4(e4, f, signe4)
    { sum the effort at a one junction }
    signe1*e1+signe2*e2+signe3*e3+signe4*e4=0.0
end

model class Zero3Junctions
    cut T1 (e, f1, signf1)
    cut T2 (e, f2, signf2)
    cut T3 (e, f3, signf3)
    { sum the flows at the zero junction }
    signf1*f1 + signf2*f2 + signf3*f3 = 0.0
end
```

In the new *bond* **model class**, the interface is:

```
cut bin(e,f,signIn)
cut bout(e,f,signOut)
```

This interface simply directly connects, or shorts, the signals e and f at its two connecting points. Hence, the *bond* **model class** performs no processing on these signals and leaves them unaltered. However the *bond* class does set the connection variables $signIn$ and $signOut$ to -1 and 1 respectively. This is so that the junctions, which the *bond* connects, are able to determine the proper polarities of the effort and flow signals. This same polarity information was previously encoded into the **cut's** through variable.

This method of implementing bond graph models works well. Because of the $signIn$ and $signOut$ connection variables, the number of equations generated by the Dymola compiler was 25. This is only slightly higher than the 15 equations generated for the solution technique used in section 3.5. Because the 0 and 1 $junctions$ were not alternated, precious screen real estate was spared. This can be especially important as the **model** or **model class** becomes more complex. However, it is often convenient to alternate junction types anyway so that flow meters and effort meters can be used to measure flow and effort values. A $flow$ meter can be encoded as:

```
model class (BGComponent) FM
    output flow
    e = 0
    flow = f
end
```

while an $effort$ meter can be encoded as:

```
model class (BGComponent) EM
    output effort
    f = 0
    effort = e
end
```

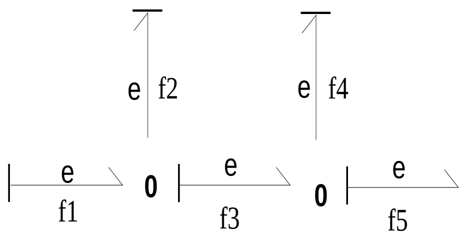

Figure 3.11. Cascading Zero Junctions

Flow meters are connected to 1-*junctions* while *effort* meters are connected to 0-*junctions*. In the BJT model, both *flow* meters and *effort* meters are used everywhere.

In this section, the goal of encapsulating the 0 and 1 *junction* information into both the *BGComponent* and *junction* **model classes** was achieved. Additionally, because the *effort* and *flow* variables are never reversed, the equations that the Dymola compiler generates (for debugging purposes) are homogeneous in either *effort* or *flow*. This makes debugging a lot easier than when using the *simpletic gyrator* model to implement bond graph models. The down side to this approach is that you need to have many different types of 0 and 1 *junctions*. That is, instead of having a single 0 or 1 *junction* which can have an infinite number of connections, the 0 or 1 *junction* presented in this section only allows for a discrete number of connections. However, junctions may be connected together to form larger ones. This is illustrated in figure 3.11.

3.8 Thermodynamic bond graphs

Modeling programs like BBSPICE have the ability to predict semiconductor device operation at different temperatures. In order to do this, the model's equations must be functions of temperature as well as for voltage and current. The equation for a resistor, for example, follows Matthiessen's rule [11]:

$$\rho = \rho_r + \rho_T \tag{3.7}$$

where the term ρ_r is the resistance that is independent of temperature. This equation is further written as:

$$\rho = \rho_{RT} \left[1.0 + \alpha_R(T - T_R) + ...\right] \tag{3.8}$$

In this equation, ρ_{RT} is the reference resistivity with respect to room temperature while α_R is known as the temperature coefficient of resistivity. For pure metals, α_R is approximately $\frac{0.004}{^\circ C}$ [11]. From this formula, a temperature sensitive resistor model can be implemented as:

```
model class (BGComponent) Resistor
    bf cut (e,f)
    parameter R = 1.0
    parameter AREA = 1.0
    parameter TR1 = 0.005
    parameter TR2 = 0.0005
    parameter TNOM = 298.15
    e = f*R*(1.0 + TR1*(T-TNOM) + TR2*(T-TNOM)*(T-TNOM))/AREA
end
```

where the variable T is the temperature of the resistor.

In the SPICE BJT model, the ohmic losses −− which occur in the collector and the emitter bulk regions−− are modeled with this resistance model.

By making each resistance and capacitance sensitive to the simulation's temperature, a more accurate simulation may be accomplished.

The one thing that this model ignores is the power that the resistor dissipates. Hence, the temperature during the simulation remains constant since the dissipated power does not turn into heat which causes the resistor's temperature to increase.

In order to actually allow the device being simulated to heat up, the model must keep track of the *entropy flow* which is generated. The resistor model can be adapted easily:

```
model class (Resistor) RS
    cut (T,sdot)
    e*f = sdot*T
end
```

In the modified resistor model (RS), an additional **cut** was added to the resistor. Hence, there is one connection for the electrical side and one connection for the thermal side of the model. The equation:

$$e * f = sdot * T \tag{3.9}$$

is used to state that all of the power that is dissipated by the resistor is turned into thermal energy. The variable T represents temperature while the variable $sdot$ represents entropy flow. It is important to note that T has the units of Kelvin. Hence, T is always positive. This makes the entropy flow ($sdot$) positive too because the product $e * f$ is always non-negative in resistive elements. The units for entropy flow are: $\frac{Joules}{^\circ K}$. Therefore, the product of T and $sdot$ is power and this is what is required.

Once the electrical energy is converted into thermal energy, several processes can occur in the thermal domain. The references [4, 13] show thermal convection, thermal radiation, and thermal conduction. In this paper, a thermal capacitor is used to store heat. If multiple thermal capacitors were used along with thermal resistors, then thermal conduction could be modeled. This was not done.

3.9 The Thermal Capacitor

The thermal capacitor has the following model:

```
model class (BGComponent) mC
    parameter gamma = 1.0
    local C
    C = gamma/e
    $C * \frac{de}{dt} = f$
end
```

The parameter gamma is called the *thermal capacitance*, and it is calculated using the following formula:

$$\gamma = c * \rho * V \tag{3.10}$$

In equation 3.10, c is the specific thermal capacitance for a given material while the product $\rho * V$ is the *mass* of the material since ρ is the density of the material and V is the volume of the material.

In [4], Cellier notes that the *modulation* of the capacitance, in the thermal capacitor, is "rather dubious". That is, how can one be sure that the thermal capacitor *always* stores energy and *never* dissipates it? The proof starts with the following equation:

$$E(t) = \int_0^t P(\tau)d\tau = \int_0^t e(\tau) * f(\tau)d\tau \tag{3.11}$$

By integrating the flow, sdot, the *charge* can be calculated as:

$$q(t) = \int_0^t f(\tau)d\tau \tag{3.12}$$

Hence, equation 3.11 can be rewritten using 3.12 as:

$$\begin{aligned} E(t) &= \int_0^t e(\tau) * \dot{q}(\tau)d\tau && (3.13) \\ q &= C * v && (3.14) \\ E(t) &= \int_0^q \frac{q}{C}dq = \int_0^q e(q)dq && (3.15) \end{aligned}$$

The modulated capacitor can be shown to always store energy because

$$\frac{dq}{dt} = f_C(t) = C * \frac{dv}{dt} = C * \dot{e}_C(t) = \frac{\gamma}{e_C(t)} * \dot{e}_C(t) \tag{3.16}$$

If equation 3.16 is used to find the *flow* of current through the capacitor, then the capacitors charge can be calculated as:

$$q_C(t) = \int_0^t f_C(\tau)d\tau = \gamma \int_0^t \frac{\dot{e}_C}{e_C(\tau)}d\tau = \gamma * log(e_C) \tag{3.17}$$

Hence, equation 3.17 shows that the capacitive charge is indeed a nonlinear function of the effort e_C, and the capacitive nature of the modulated capacitor has thus been verified because equation 3.15 remains valid when equation 3.17 is substituted into it.

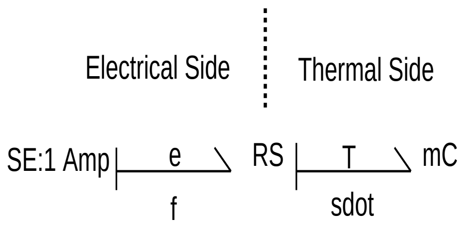

FIGURE 3.12. Small Thermal Circuit

3.10 A Small Thermal Circuit

Figure 3.12 shows an example of a bond graph model which includes both electrical and thermal parts.

3.11 Things to be Careful About

There are some important things to remember when the modeler creates **model classes** that use inheritance. When a model is entered in by hand, it is the responsibility of the modeler to handle the $name - space$ of the model's variables. The **models** that were shown in this chapter have had lots of *bonds* and the name-space of the *bonds* was simply $b1$, $b2$, $b3$, etc... In the *Dymo Draw* environment, the management of these names is done automatically. The most obvious case of this automatic management is when the modeler duplicates an object. The name of the new object is based off the name of the old object. If there is no inheritance used, then the management is trivial since a numbering scheme can be used to create unique names for each object. However, once inheritance is introduced, the task becomes quite complex. When a **model class** is read into *Dymo Draw*, it is possible for *Dymo Draw* to figure out what objects make up that particular **model class** as well as what objects make up its base classes. However, the *Dymo Draw* environment does not know what objects are declared in **model classes** which use the current **model class** as its own base class. That is, inheritance can be modeled by a singularly linked list and not a double linked list. Hence, the problem of having a non-unique name space for objects within an inheritance hierarchy can occur because the names of the objects which are contained in more derived classes may not be accounted for. If there is not a unique name for each object, then the Dymola compiler will find that certain variables are over-determined because the objects with the same names generate variables of the same name.

Another important thing to consider when designing **model classes** with inheritance is that the Dymola compiler only allows the modeler to make connections to a connector at one level in an inheritance hierarchy. Hence, if the *simpletic gyrator* implementation is used to implement a bond graph model, then the modeler can only make connections to any given 0 or 1-*junction* in one layer of the inheritance hierarchy. By using the 0 and 1 *junction* **model classes** that have multiple distinct connection points (section 3.7), the modeler may then make connections in multiple layers of the inheritance hierarchy.

Chapter 4
The BJT Bond Graph Model

4.1 Introduction

The objective of this chapter is to convert the transistor model found in [4, 7] into a bond graph. Additionally, by using the techniques outlined in [7], this bond graph was specialized to create both vertical, figure 4.1a, and lateral, figure 4.1b, NPN and PNP transistor models.

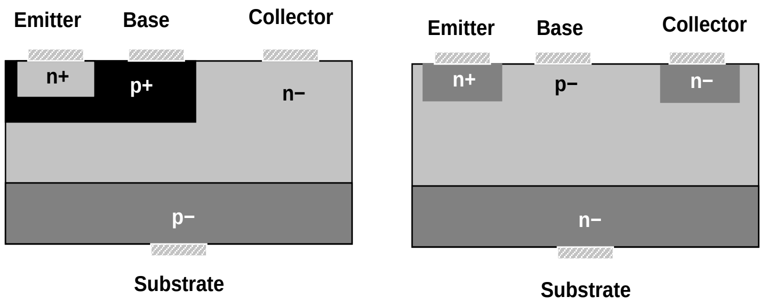

Figure 4.1. Vertical and Lateral NPN Transistors

4.2 Corrections to the BJT model

The primary source for the BJT model was [7]. Figure 4.2 shows the circuit model that is found in [7]. This model includes three resistors, three diodes, and two current sources. The resistors, labeled RC_{Int} and RE_{Int}, account for the ohmic losses which occur in the bulk regions of a transistor. The base resistance, rbb, models the resistance in the base. The diodes, labeled dbc and dbe, model the transistor action (equation 4.8) of the transistor.

By examining figure 4.2 and using the equations provided by [7], a small error was discovered in the model's equations. This error can be found by applying Kirchoff's

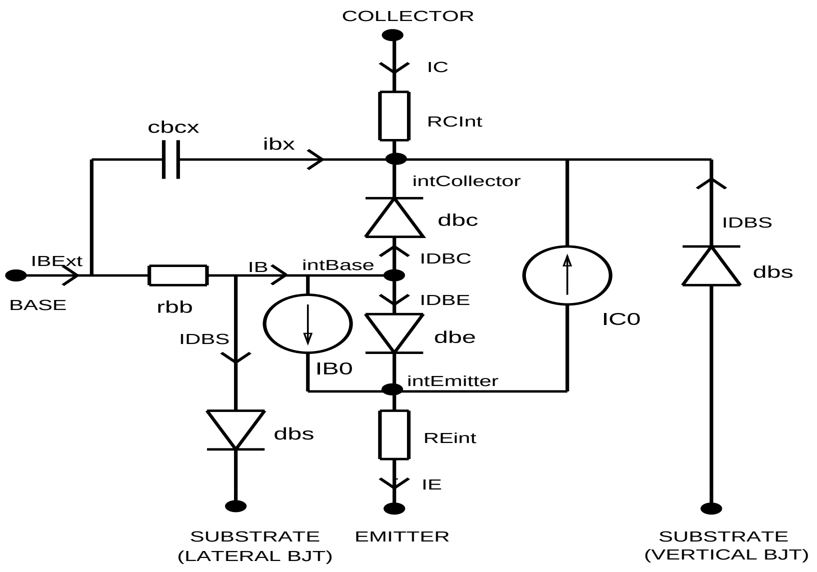

FIGURE 4.2. Electric Circuit Model of NPN Transistor

current law at the base. The summation of the currents at *intBase* produces the following result:

$$-i_{base} + i_{dbc} + i_{dbe} + I_{B0} = 0 \tag{4.1}$$

In equation 4.1, i_{base} represents the current contribution from the substrate diode (*dbs*) and the base resistance (*rbb*). Additionally, i_{DBC} is the current $J_S * exp(\frac{qV_{BC}}{kT})$ and i_{DBE} is the current $J_S * exp(\frac{qV_{BE}}{kT})$. In order to simplify this equation, the current $IB0$ [7] :

$$IB0 = \frac{i_{DBC}}{BFv} + \frac{i_{DBE}}{BRv} + i_{en} + i_{cn} - i_{DBE} \tag{4.2}$$

can be used to obtain the result:

$$-i_{base} + i_{DBC} + \left(\frac{i_{DBE}}{BFv} + \frac{i_{DBC}}{BRv} + i_{en} + i_{cn}\right) = 0 \tag{4.3}$$

Therefore, the value of i_{base} can be further written as:

$$i_{base} = i_{DBC} + \left(\frac{i_{DBE}}{BFv} + \frac{i_{DBC}}{BRv} + i_{en} + i_{cn}\right) \tag{4.4}$$

If i_{loss} is defined as:

$$i_{loss} = \left(\frac{i_{DBE}}{BFv} + \frac{i_{DBC}}{BRv} + i_{en} + i_{cn} \right) \tag{4.5}$$

then equation 4.4 can be written concisely as:

$$i_{base} = i_{DBC} + i_{loss} \tag{4.6}$$

where i_{loss} is the main current component in the active region (figure 4.3).

As a sanity check, equation 4.6 can be used to calculate the current gain β_0 [5] :

$$\beta_0 = \frac{I_C}{i_{base}} = \frac{I_C}{i_{DBC} + i_{loss}} \tag{4.7}$$

where I_C has the definition found in [9] :

$$J_n = J_s \left[exp \left(\frac{qV_{BC}}{kT} \right) - exp \left(\frac{qV_{BE}}{kT} \right) \right] \tag{4.8}$$

where the term J_S is:

$$J_S = \frac{q^2 n_i^2 D_n}{Q_n} \tag{4.9}$$

and n_i is the intrinsic concentration of electronics, D_n is the diffusion constant, and Q_n is the space charge.

By inspection, equation 4.7 predicts that a lossless transistor operating in the *reverse-active* region (figure 4.3) would have a current gain of 1.0. However, the theoretical gain should be β_R.

As noted in [12], the transistor model is not simply two diodes, *dbc* and *dbe*, in series. If this were true, then the current predicted by equation 4.8 would only flow through a transistor operating in the *forward-active* region (figure 4.3) if the *reversed-biased* diode, *dbc*, had a large reverse bias voltage applied across it. However, the base-collector diode (*dbc*) in a transistor is usually not subjected to such a voltage. Instead, the base layer is made thin enough so that it allows a large current to flow through it without the requirement that the base-collector diode operate in its

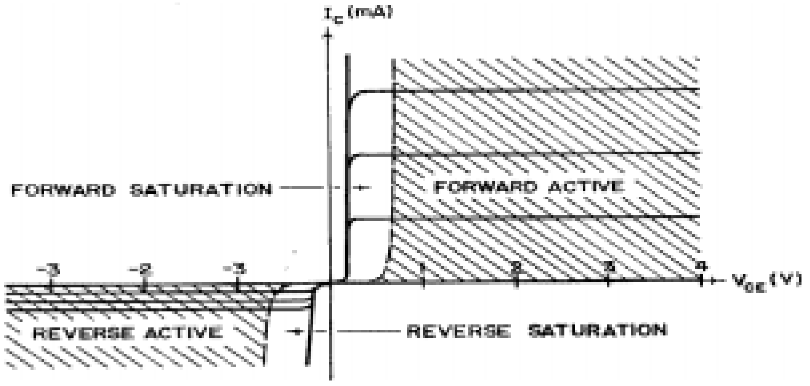

FIGURE 4.3. IV Curve for NPN Transistor [14]

breakdown region. Hence, the base-collector diode only causes a small leakage current to flow. The results presented in [7] used a transistor biased in the *forward-active* and *forward-saturation* regions. Hence, this explains why the error discussed in this section was not detected in [7] since the error was not significant in these regions. This error is fixed in the bond graph model of the BJT

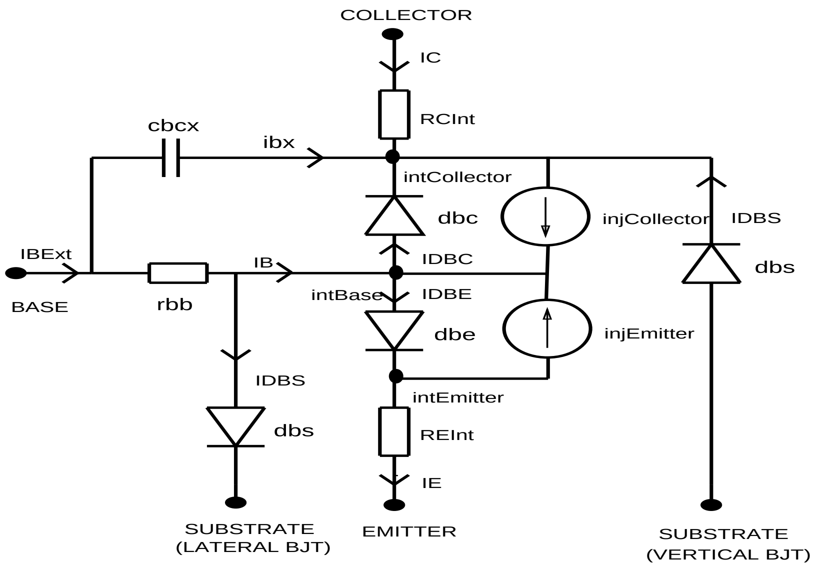

FIGURE 4.4. Modified Hild BJT Model

While making this correction, the circuit model in figure 4.2 was slightly modified. This new circuit model is shown in figure 4.4. In the revised model, the losses shown in equation 4.5 are included in the *dbc* and *dbe* diode models themselves. Finally, the two current sources are different. The current sources in figure 4.4 are associated with the injection currents described by [9]. These currents are:

$$injCollector = J_s exp\left(\frac{qV_{BE}}{kT}\right) = i_{DBE} \tag{4.10}$$

which represents the base-emitter diode current that is injected into the collector, and:

$$injEmitter = J_s exp\left(\frac{qV_{BC}}{kT}\right) = i_{DBC} \tag{4.11}$$

which represents the collector-base diode current that is injected into the emitter. If Kirchoff's law is now applied at intBase in figure 4.4, the value for i_{base} is now equal to i_{loss}, which is expected. Additionally, if Kirchoff's law is applied at either *intCollector* or *intEmitter*, the current calculated is the current predicted by equation 4.8.

4.3 Creating a BJT Bond Graph Model

The bond graph model of the BJT circuit is shown in figure 4.5. The conversion is straight forward using the principles outlined in 3.2, 3.3, and 3.8.

4.4 Some of the Dymola Code for the BJT

This section presents some of the Dymola code which was used to model the BJT bond graph in figure 4.5. Because of the size of the source code that was needed to glue these parts together into NPN and PNP transistors, only a high level description of how these **model classes** were constructed is given. The full source code, however, is available on the world wide web at http://www.ece.arizona.edu/c̃ellier/ms.html. In [4, 7], the authors use a Dymola coding style which uses a more complex modeling notation than the *Dymo Draw* modeling tool likes to use. This is because *DymoDraw* does not have to worry about carnal tunnel syndrome, and hence, the

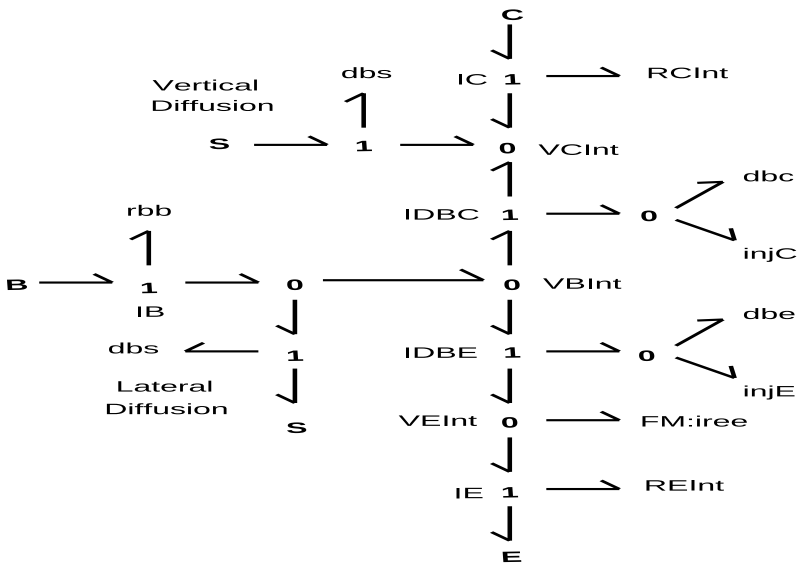

FIGURE 4.5. Modified Hild BJT Model As a Bond Graph

program uses a verbose, but simple, modeling notation that is easier for the computer to manipulate.

The most important parts of the transistor model are the diodes. The base class, which was used by the three different diode **model classes** is:

```
model class BGJDiode

    output Ic { Capacitor Current }
    output Ir { Recombination Current }
    output Id { Diode Current }
    output I1 { Recombination Current }
    output cdep { Depletion Capacitance }
    output cdif { Diffusion Capacitance }
    output pdiode { Power Dissipation = e*Id }

    local power

    cut B(e,f,.) {Cut for Electrical Side }
    cut B1(T,sdot,.) { Cut for Thermal Side }

    { the diode and the losses are dissipated as heat }

    power = (Id + Ir + I1)*e

    { electrical-thermal power balance equation }
    power = T*sdot

    { the current flowing through the diode is: }

    f = (Ic + Ir + Id + I1)

    { The current flowing through the capacitors is: }

    Ic = der(e)*(cdep + cdif)

    pdiode = Id*e
end
```

The **model class** *BGJDiode* is considered to be abstract since it is not usable by itself. The *BGJDiode* class accounts for three different current components: Ic is the current that flows through the diffusion (*cdif*) and depletion region (*cdep*) capacitors; Ir and I1 are the currents which model the loss due to recombination effects, and Id is the non-linear diode current $J_S * exp(q * \frac{V}{kT})$ which flows through a forward biased diode. The diode is modeled with the following circuit model:

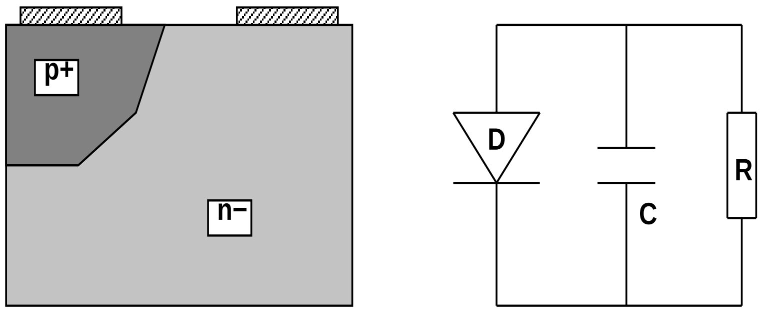

FIGURE 4.6. Simple Diode Circuit Model

The three diode **model classes** which use this base class are: The substrate diode *BondDBS*, the base-collector diode *BondDBC*, and the base-emitter diode *BondDBE*:

```
model class (BGJDiode) BondDBS
    parameter AREA = 1.0
    parameter NS = 1.0

    local vtns
    local ise
    local denom

    terminal idiode

    { cut B2 is used to calculate the listed across variables }

    cut B2 (T, Vt, GMINDCv, CJSv, MJSv, VJSv, dummy3, dummy4, ISSv, dummy5)

    vtns = Vt*NS

    ise = ISSv*exp(e/vtns)

    idiode = (ise - ISSv)*AREA + GMINDCv*e

    Id = idiode
    Ir = 0.0 {no losses}
    I1 = 0.0 {no losses}

    cdep = AREA*CJSv/denom
    cdif = 0.0

    denom = (1.0 - e/VJSv)**MJSv
end

model class (BGJDiode) BondDBC

    parameter AREA = 1.0
    parameter NR = 1.0
    parameter NC = 1.0
    parameter XCJC = 1.0

    terminal idiode { ideal diode current }

    local ise
    local vtnr vtnc
    local denom
    local XCJCv

    external qb { base charge citation:?? }

    cut B2 (T, Vt, GMINDCv, CJCv, MJCv, VJCv, TRv, BRv, ISv, ISCv)

    idiode = (ise - ISv)*AREA + GMINDCv*e

    vtnr = Vt*NR
    vtnc = Vt*NC

    ise = ISv*exp(e/vtnr)
```

```
        Ir = AREA*ISCv*(exp(e/vtnc) - 1.0)
        Id = idiode/qb
        I1 = idiode/BRv

        XCJCv = XCJC

        denom = (1.0 - (e - 0.5*Vt*exp((e-VJCv)/Vt))/VJCv)**MJCv
        cdep = XCJCv*AREA*CJCv/denom
        cdif = TRv* (ise/vtnr + GMINDCv)
    end

    model class (BGJDiode) BondDBE
        parameter AREA = 1.0
        parameter NF = 1.0
        parameter NE = 1.0

        local vtnf vtne
        local ise
        local denom

        external qb dqb

        terminal idiode

        cut B2(T, Vt, GMINDCv, CJEv, MJEv, VJEv, TFv, BFv, ISv, ISEv)

        vtnf = Vt*NF
        vtne = Vt*NE

        ise = ISv*exp(e/vtnf)

        idiode = (ise - ISv)*AREA + GMINDCv*e

        Ir = AREA*ISEv*(exp(e/vtne) - 1.0)
        Id = idiode/qb
        I1 = idiode/BFv

        denom = (1.0 - (e - 0.5*Vt*exp((e-VJEv)/Vt))/VJEv)**MJEv

        cdep = AREA*CJEv/denom
        cdif = TFv*((ise/vtnf + GMINDCv)/qb - Id*dqb)/(qb*qb)
    end
```

The base-collector (*dbc*) and base-emitter (*dbe*) diodes act as a non-linear resistor pair [14]. Figure 4.7 shows the two ideal BJT models which are used to explain this high level effect [14]. Figure 4.7.(A) shows the BJT when the base-emitter diode is forward-biased and the base-collector diode is reverse-biased. If Ohm's law is used to think about the resistance of a diode, then:

$$R_{diode} = \frac{V_{Diode}}{J_S * exp(\frac{q*V_{Diode}}{kT})} \tag{4.12}$$

and equation 4.12 can be used to imply that the resistance of the base-collector diode is very large (since the denominator is very small) while the resistance of the base-emitter diode is much closer to 0 since the denominator of 4.12 becomes larger due to the application of a forward-bias voltage. In figure 4.7, this is represented by labeling the emitter region with an N^- and labeling the collector region with N^+. The N^+ region indicates that the collector region has characteristics closer to wire, while the N^- indicates that the emitter region has characteristics which are closer to a resistor. As the voltage across the base-emitter diode is increased, a high-level effect called quasi-saturation becomes dominant. Quasi-saturation is the condition whereby the base-collector diode becomes forward-biased even though the external biasing circuitry is applying a reverse-bias. Figure 4.7.(B) represents quasi-saturation by indicating that the emitter region attains a minimum resistance value (the denominator of 4.12 is at a maximum) and that the collector region is starting to look more like a resistor. From a power standpoint, quasi-saturation occurs when the BJT is operated in the *forward-saturation* or the *reverse-saturation* regions of figure 4.3. In these regions, the current flowing through the transistor is no longer controlled by the bias voltages V_{BC} and V_{BE} but instead by the power supply and the external circuitry.

The collector and the emitter ohmic losses (RC_{Int} and RE_{Int}) are modeled with the RS **model class** *BondVR*:

```
model class BGComponentRS

    cut B(e,f,.) { Electrical Side }
    cut B1(T,sdot,.) { Thermal Side }

    parameter TNOM = 298.15

    output power
```

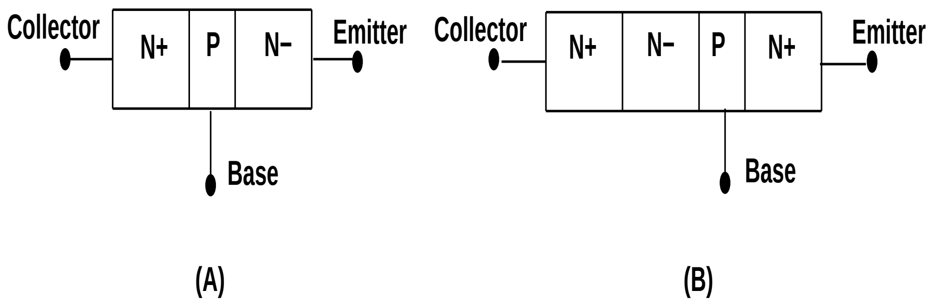

FIGURE 4.7. Ideal Quasi-Saturation BJT Model

```
    local DTemp DTempSq

    DTemp = (T - TNOM)
    DTempSq = DTemp*DTemp

    power = e*f
    T*sdot = power
end

model class (BGComponentRS) BondVR

    parameter AREA = 1.0
    parameter R
    parameter TR1
    parameter TR2

    e = R*(1.0 + TR1*DTemp + TR2*DTempSq)*f/AREA
end
```

The base resistance (*rbb*) has a more complex model than either of the ohmic losses which occur in the collector or the emitter bulk regions:

```
model class (BGComponentRS) BondRBB

    parameter AREA=1.0
    parameter RB RBM
    parameter TRB1 TRB2
    parameter TRM1 TRM2

    local RBv RBMv
    local r

    external qb
    RBv = RB*(1.0 + TRB1*DTemp + TRB2*DTempSq)
    RBMv = RBM*(1.0 + TRM1*DTemp + TRM2*DTempSq)

    r = (RBMv + (RBv - RBMv)/qb)/AREA
    e = r*f
end
```

Some of the **model classes** use an external variable called **qb**. The variable **qb** represents the charge stored in the base and it is used, in part, to model the flatness of the *foward-active* and *reverse-active* IV curve shown in figure 4.3. Qb is also used to predict the fall off of this IV curve in the *forward-saturation* and the *reverse-saturation* regions of transistor operation.

The last **model class** that is needed is the current source which is used for the injection currents shown in equations 4.11 and 4.10 :

```
model class (BGComponent) BondMFSource
    output power
    input ix
    f = -ix
    power = e*f
end
```

4.5 The BJT Object Hierarchy

Figure 4.5 shows the inheritance model that was used for the construction of the BJT bond graphs. The hierarchy contains 3 abstract **model classes**. The *BJTBase* **model class** constructs the bond graph model shown in figure 4.5. However, this **model class** does not include the bonds which connect the 1-*junction* labeled I_{DBC} to the 0-*junctions* labeled V_{BInt} and V_{CInt}. It also does not include the bonds which connect the substrate diode from the substrate material to the 0-*junctions* labeled VC_{Int} or VB_{Int}. It should be noted that figure 4.5 shows two substrate diodes. Only one diode is needed. The other one is represented by a current source (*BondMFSource*). This current source generates no current flow, and hence, it does not contribute to the model's power flows.

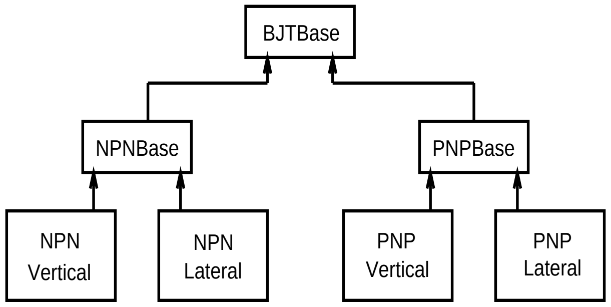

FIGURE 4.8. BJT Inheritance Diagram

The second level of the inheritance hierarchy contains the $NPNBase$ and $PNPBase$ **model classes**. The $NPNBase$ **model class** instantiates two sets of bonds. The first set connects the 0-*junction* VB_{Int} to the 1-*junction* I_{DBC} and the 1-*junction* I_{DBC} to the 0-*junction* VC_{Int}. Likewise, the second set connects the 0-*junction* VB_{Int} to the 1-*junction* I_{DBE} and the 1-*junction* I_{DBE} to the 0-*junction* VE_{Int}. The end result is that the base-collector diode (*dbc*) is connected from the base to collector and the base-emitter diode (*dbe*) is connected from the base to the emitter. The $PNPBase$ model, on the other hand, does just the opposite. These classes are also considered to be abstract since the substrate diode is still not connected.

The last level of the hierarchy constructs four concrete classes. The NPN Vertical and the PNP Vertical **model classes** connect the substrate diode from the substrate material to the collector while the NPN Lateral and the PNP Lateral **model classes** connect the substrate diode from the base to the substrate material.

The final step is to create transistor models that have the desired parameters. Using the $Has{-}A$ design technique, an NPN Vertical transistor can be created as:

```
model class (NPNCuts) NPNV2
    parameter ISC = 0.1E-6 ISE = 0.0
    parameter NC = 2.0 NE = 1.5
    parameter BF = 100 BR = 1
    parameter VAF = 1.0E+30 VAR = 1.0E+30
    parameter IKF = 1.0E+30 IKR = 1.0E+30
    parameter NR = 1.0 NF = 1.0 NS = 1.0
    parameter IS = 0.11E-9 ISS = 0.11E-9
    parameter TR = 1.0E-12 TF = 1.0E-12
    parameter VJC = 0.75 CJC = 3.6E-12 MJC = 0.33
    parameter VJE = 0.75 CJE = 5.7E-12 MJE = 0.33
    parameter VJS = 0.75 CJS = 11.0E-12 MJS = 0.33
    parameter XCJC = 1.0
    parameter RB = 200.0 TRB1 = 0.00 TRB2 = 0.000
    parameter RBM = 100.0 TRM1 = 0.00 TRM2 = 0.000
    parameter IRB = 0.0
    parameter RC = 750.0 TRC1 = 0.00 TRC2 = 0.000
    parameter RE = 123.3 TRE1 = 0.00 TRE2 = 0.000
    parameter XTI = 3.0 XTB = 0.0
    parameter GMINDC= 1.0E-12
    parameter TNOM = 298.15
    parameter EG = 1.16
    parameter AREA = 1.0

    submodel (NPNV) NPN
        (ISC=ISC, ISE=ISE, NC=NC, NE=NE, BF=BF, BR=BR,
        VAF=VAF, VAR=VAR, IKF=IKF, IKR=IKR,
        NR=NR, NF=NF, NS=NS,
        IS=IS, ISS=ISS, TR=TR, TF=TF,
        VJC=VJC, CJC=CJC, MJC=MJC,
        VJE=VJE, CJE=CJE, MJE=MJE,
        VJS=VJS, CJS=CJS, MJS=MJS,
        XCJC=XCJC,
        TRB1=TRB1, TRB2=TRB2, RB=RB, RBM=RBM,
        TRM1=TRM1, TRM2=TRM2, IRB=IRB,
        TRC1=TRC1, TRC2=TRC2, RC=RC,
        TRE1=TRE1, TRE2=TRE2, RE=RE,
        XTI=XTI, XTB=XTB,
        GMINDC=GMINDC, TNOM=TNOM, EG=EG, AREA=AREA)

    {make connections to the actual Vertical BJT :}

    connect NPN:S at S
    connect NPN:B at B
    connect NPN:E at E
    connect NPN:C at C
    connect NPN:T at T
end
```

This **model class** ($NPNV2$) provides similar functionality to the $SPICE$ **.MODEL** command which allows the modeler to create a library of NPN or PNP transistors which use the parameter values that the modeler prefers.

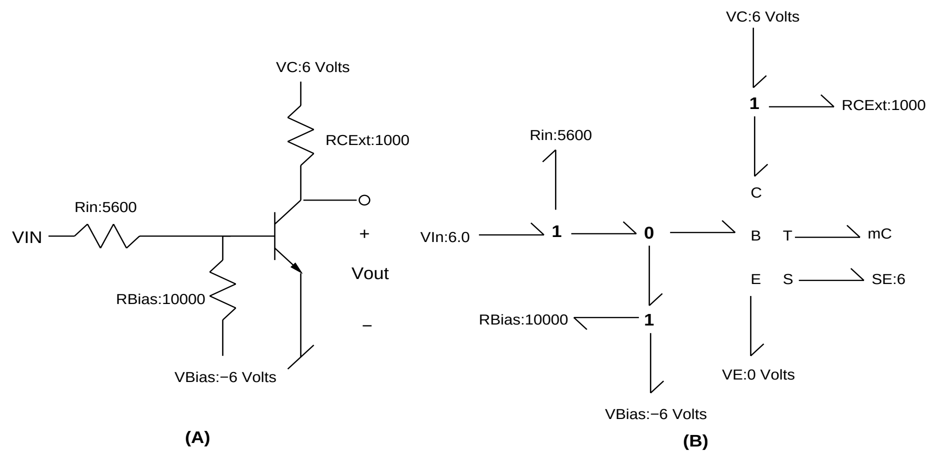

FIGURE 4.9. NPN Inverter

4.6 Simulation of a BJT Inverter

The same NPN Inverter circuit that was used in [7] was also used in this thesis. This was done to take advantage of an established benchmark. The NPN inverter circuit diagram is shown in figure 4.9.(A) while the NPN inverter bond graph model is shown in figure 4.9.(B). The input to the circuit was a 0 to 6 to 0 volt pulse. A presentation of the simulation's results are shown in figures 4.10 and 4.11. The results of the simulation show that the BJT bond graph model does not provide an intuitive interpretation of the power flow since the emitter current source generates power. The amount of power that is generated is shown in figure 4.11. The formula used to generate this plot is:

$$P_{diode} = V_{BC} * J_S * exp(\frac{qV_{BC}}{kT}) - V_{BE} * injEmitter \quad (4.13)$$

$$P_{diode} = J_S * exp(\frac{qV_{BC}}{kT}) * (V_{BC} - V_{BE}) \quad (4.14)$$

$$P_{diode} = -J_S * exp(\frac{qV_{BC}}{kT}) * (V_{CB} + V_{BE}) \quad (4.15)$$

By defining the collector-emitter voltage as:

$$V_{CE} = V_{CB} + V_{BE} \quad (4.16)$$

equation 4.15 can be simplified to:

$$P_{diode} \quad = \quad -J_S * V_{CE} * exp(\frac{qV_{BC}}{kT}) \tag{4.17}$$

The interpretation of this formula, with reference to figure 4.11, is that the current source dissipates more power than the base-collector diode (*dbc*) supplies to it. Hence, the straight forward conversion of the circuit shown in 4.9.(A) to the bond graph model shown in figure 4.9.(B) is not very beneficial since it does not provide a bond graph model which interprets the power flow through a transistor well.

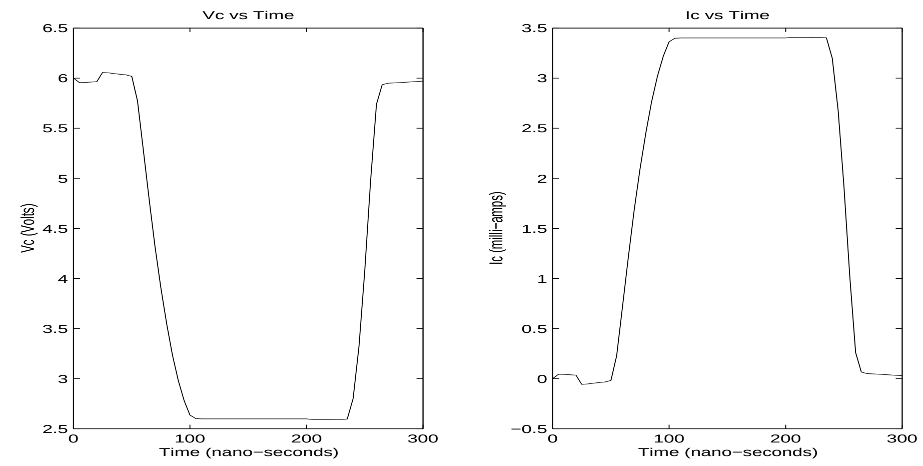

FIGURE 4.10. Transient Response of Inverter Circuit 4.9

4.7 A Revised BJT Bond Graph Model

The bond graph model shown in figure 4.12 uses the fact that the power dissipated across the transistor is:

$$P_{diode} = (V_{BE} - V_{BC})(i_{dbe} - i_{dbc}) \tag{4.18}$$

where the quantity $(i_{dbe} - i_{dbc})$ is the current predicted by equation 4.8. Equation 4.18 can be further simplified by considering the fact that:

$$V_{CE} = V_{CB} + V_{BE} = V_{BE} - V_{BC} \tag{4.19}$$

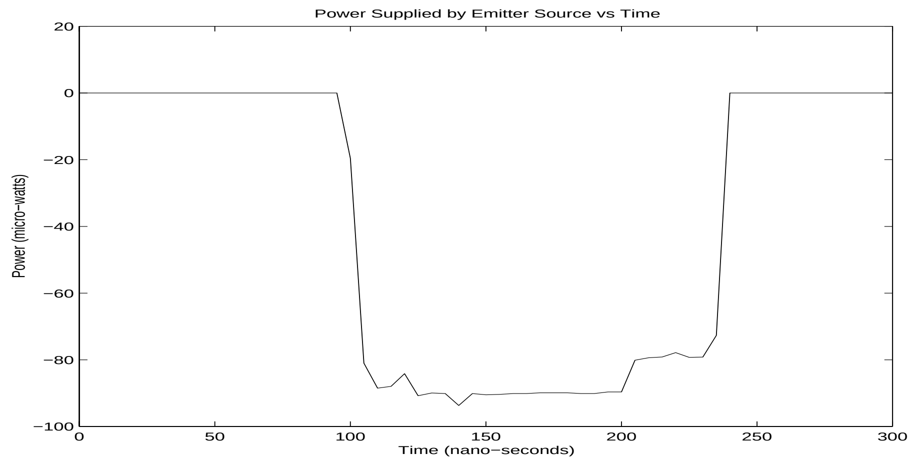

FIGURE 4.11. Plot of the Power Generated by injEmitter

since

$$V_{CB} = -V_{BC} \tag{4.20}$$

By using substituting these equations into equation 4.18, the power dissipated by a transistor can be calculated as:

$$P_{diode} \quad = \quad V_{CE} * (i_{dbe} - i_{dbc}) \tag{4.21}$$

The final result (equation 4.21) shows that the current that flows through V_{CE} is the superposition of the currents that flow through a $forward$-$active$ and a $reverse$-$active$ biased NPN transistor circuit.

This result was applied to create the bond graph model shown in figure 4.12. The current sources, $injCollect$ and $injEmitter$, are gone and they are no longer part of the bond graph BJT model. Instead, a new component was created and it is shown with the label RCE. RCE is called a *word bond graph* element because the balance of power has the form:

$$P_{diode} = \sum_{i=0}^{n} e_i * f_i = \sum_{i=0}^{n} p_i = 0 \tag{4.22}$$

This formulation of power conservation is different than the equations which are used for 0 or 1-*junctions* (equation 3.3) since neither $effort$ (e) nor $flow$ (f) is constant.

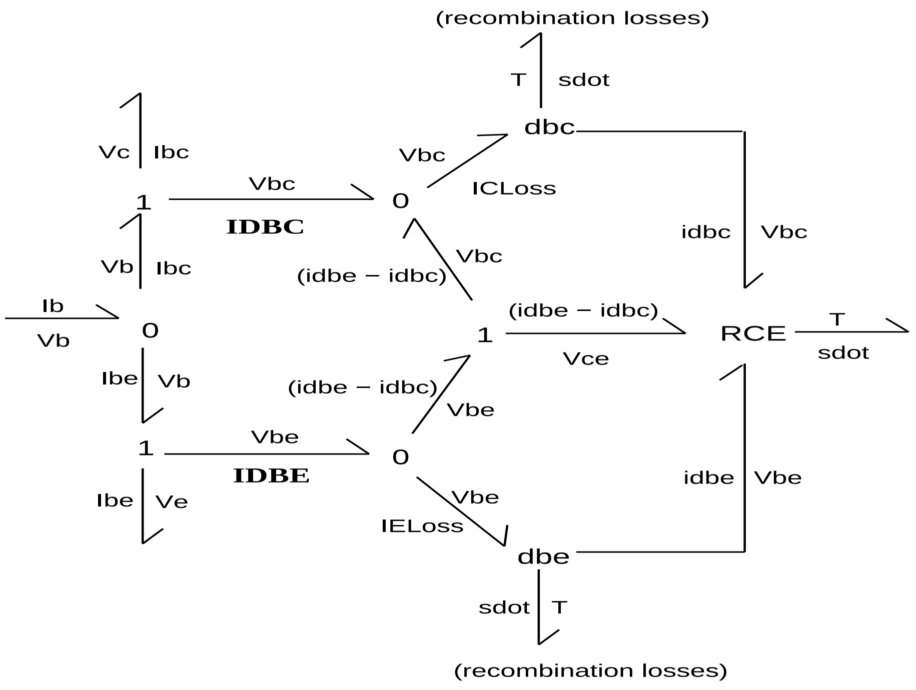

FIGURE 4.12. A Second BJT Bond Graph Model

Hence, equation 4.18 simply represents the sum of four partial powers which are:

$$P_{diode} = V_{BE} * i_{dbe} - V_{BE} * i_{dbc} - V_{BC} * i_{dbe} + V_{BC} * i_{dbc} \tag{4.23}$$

By using the result of the derivation (equation 4.21), equation 4.23 can be interpreted as the resistance of the transistor (R_{CE}):

$$R_{CE} = \frac{V_{CE}}{i_{dbe} - i_{dbc}} = \frac{V_{CE}}{I_{CE}} \tag{4.24}$$

Figure 4.13 shows the power which is dissipated by the RCE resistor when the circuit shown in figure 4.9.(B) is simulated. Unlike the results that were presented in figure 4.11, the power which is dissipated across the transistor is now positive. Hence, the modification to the bond graph model provided a better way to represent the power dissipated across the transistor.

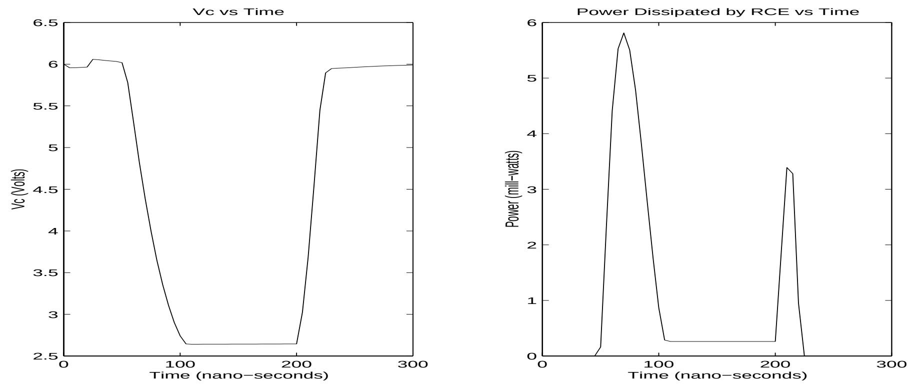

FIGURE 4.13. Power Dissipated by RCE

4.8 Thermodynamic Results

So far, only the electrical model has been discussed while the thermodynamic model was not. The thermodynamic part of the model is shown in figure 4.14. The model is very simple because all the entropy flows that are generated by the resistive elements are added up by a 0-*junction*. The **T** in figure 4.14 is the same **T** that is show in 4.9.(B).

$$\begin{tikzcd}
\mathbf{0} \arrow[r] & \mathbf{0} \arrow[r] & \mathbf{0} \arrow[r] & \mathbf{0} \arrow[r] & \mathbf{0} \arrow[r] & \mathbf{0} \arrow[r] & \mathbf{0} \arrow[r, "T", "sdot"'] & \mathbf{T} \\
dbs \arrow[u, "T", "sdot1"'] & dbc \arrow[u, "T", "sdot2"'] & dbe \arrow[u, "T", "sdot3"'] & RCE \arrow[u, "T", "sdot4"'] & rbb \arrow[u, "T", "sdot5"'] & RCInt \arrow[u, "T", "sdot6"'] & REInt \arrow[u, "T", "sdot7"'] &
\end{tikzcd}$$

FIGURE 4.14. Thermodynamic Bond Graph

In figure 4.9.(B), the thermodynamic power is connected to a thermodynamic capacitor (mC).

The value of γ (section 3.10) can be found through the calculation: $c * \rho * V$. A set of values which can be plugged into this equation are found in [15] :

$Parameter$	$Value$	$Description$
c	$0.714\ \frac{W-s}{g-^{\circ}C}$	$Specific\ Heat\ of\ Silicon$
ρ	$2.33\ \frac{g}{cm^3}$	$Density\ of\ Silicon\ Material$
V	$(10\mu m)^3$	$Volume\ of\ the\ Silicon\ Mass$

With these values, $\gamma \approx 1.664e{-}9$. The plot of the temperature is shown in figure 4.15.

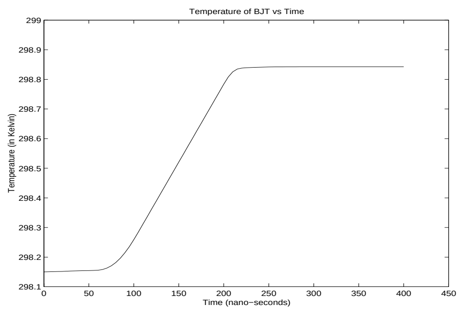

FIGURE 4.15. Temperature Rise of Semiconductor (BJT)

Chapter 5
Results

5.1 Introduction

This section presents the simulation results for the bond graph model of the NPN and PNP transistors. Three different circuits are presented. The first is an NPN inverter circuit, the second is a PNP buffer circuit, and the third is an opamp circuit. These circuits were also used as benchmarks in [7].

The BBSPICE simulation program, which is a circuit modeling program that was developed from HSPICE by the Burr Brown Corporation, is used to provide a comparison for the data produced by the bond graph BJT circuits. The BBSPICE program is able to determine, and then output, the initial values for each state variable in the model. These initial values were then used to specify the initial values for each state variable in the Dymola simulation.

All the models in this chapter are simulated for 400 nano-seconds. The input for all the models is a 180 nano-second pulse which begins 20 nano-seconds into the simulation. The input has a minimum amplitude of 0 volts and a maximum amplitude of 6 volts.

5.2 The NPN Inverter

The NPN test circuit which is simulated in this section uses a lateral NPN transistor. The particular type of circuit which is simulated is an NPN inverter circuit. The circuit is shown in figure 5.1.

The output of this circuit's simulation is shown in figure 5.2. The sub-figure on the right shows the power which is dissipated by the resistor R_{CE} in the transistor

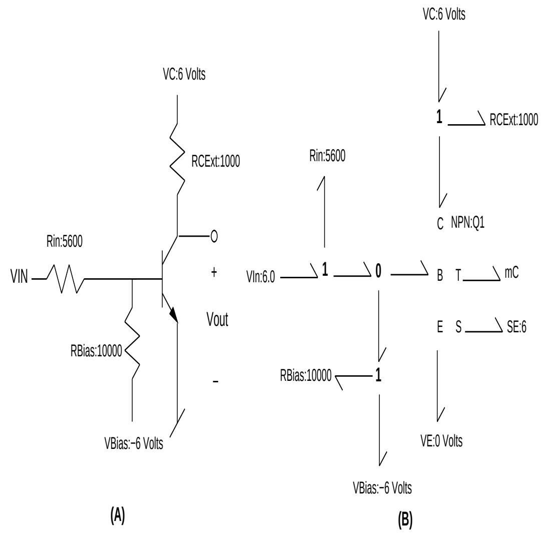

FIGURE 5.1. NPN Inverter Test Circuit

model. This figure shows that lobes are produced when the circuit is turned on and then off. At these points, the power which is dissipated across the resistance is at a maximum. The power dissipation between the lobes is zero because the circuit goes into quasi-saturation and both of the transistor's diodes are conducting currents. These currents are nearly equal and of opposite sign. Therefore, they cancel each other out.

Chapter 2 discussed how Dymola uses an equation sorter to produce a set of equations which determines the values of a set of unknown variables. The Dymola model for the NPN inverter produces the following set of equations:

Sequence of	60	equations
Sequence of	8	simultaneous equations
Sequence of	37	equations
Sequence of	5	simultaneous equations
Sequence of	127	equations

When Dymola needs to use simultaneous equations, it usually uses matrix inversion to determine a solution. Therefore, the more simultaneous equations which are needed, the slower the calculations will be.

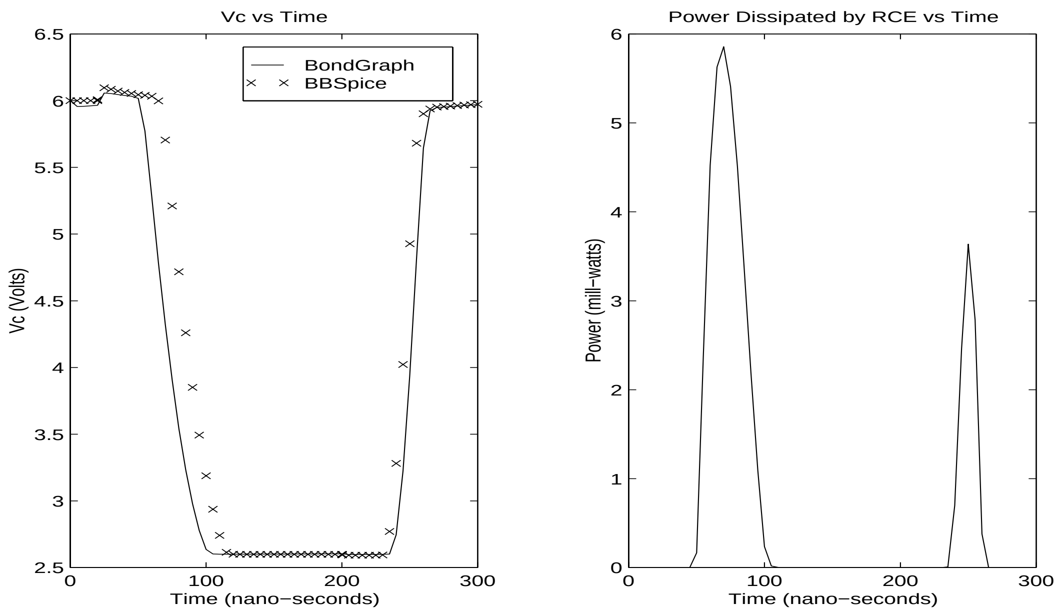

FIGURE 5.2. NPN Inverter Circuit Output (collector voltage)

Dymola allows the modeler to create Dymola scripts so that he/she can compile and simulate Dymola models with ease. The following Dymola script was used to compile and simulate the NPN inverter:

```
{ don‘t make any default connections }
set  defaultconnect  off
{ read in the model }
enter model npntst.dym
{ set the initial value of each state variable }
variable value NPNL1::NPN::dbc.e=-6.1538
variable value NPNL1::NPN::dbe.e=-0.1538
variable value NPNL1::NPN::dbs.e=-6.1538
variable value bondmc.T=298.15
{ configure the Dymola Compiler as desired }
set eliminate       on  { eliminate common expressions       }
set Evaluate        on  { evaluate constants                }
set RemoveAuxiliary on  { remove auxiliary equations        }
set SolveSymbolic   on  { solve small systems of equations  }
                        { symbolically                      }
set RemoveAuxiliary on
{ generate the model's equations                            }
partition
{ specify simulation parameters for end time, communication }
{ interval, and the integration algorithm which should be   }
{ used                                                      }
experiment StopTime       = 400.0E-9
experiment OutputInterval = 5.0E-9
experiment Algorithm      = Lsodar
output experiment
{ write the C code that implements the model }
output model
{ compile the model }
compile
{ simulate the model }
simulate
```

This simulation takes approximately 0.22 seconds to execute and this measurement does not include the time needed to compile the Dymola model.

5.3 The PNP Test Circuit

The second circuit use a vertical PNP transistor. This circuit is similar to the one simulated in the previous section. The circuit is shown in figure 5.3.

The PNP circuit always conducts current because the base voltage is below the collector voltage. Therefore, current flows from the collector into the base because

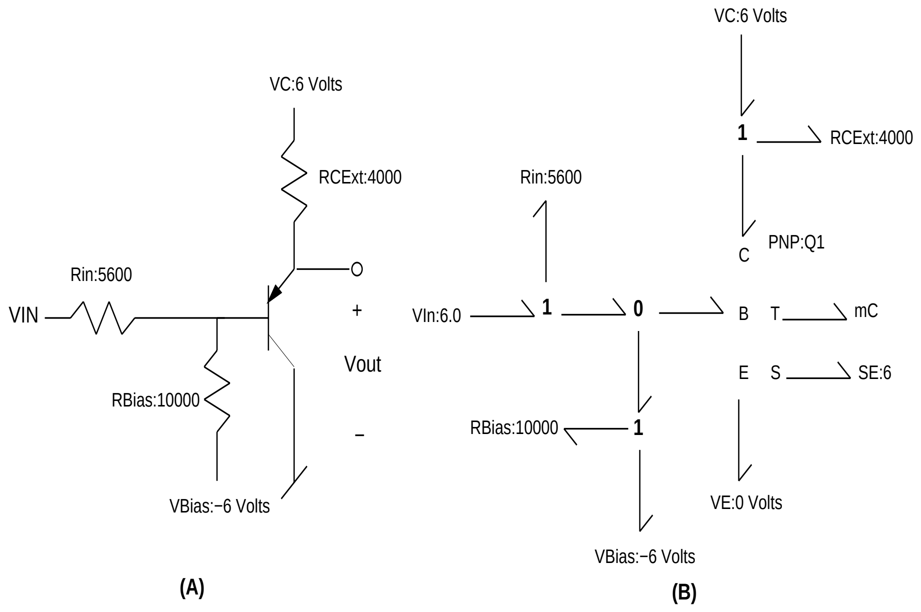

FIGURE 5.3. PNP Test Circuit

the base acts like a ground.

The Dymola model for the PNP circuit produces the following number of equations:

Sequence of	64	equations
Sequence of	8	simultaneous equations
Sequence of	69	equations
Sequence of	5	simultaneous equations
Sequence of	91	equations

and the simulation executes in approximately 0.15 seconds. This measurement does not include the time needed to compile the Dymola model.

5.4 The OpAmp Circuit

The opamp model, which is shown in figures 5.5 and 5.6, has 12 transistors, four capacitors, and one resistor. Six of these transistors are vertical NPN transistors, while the other six are lateral PNP transistors.

The simulation of the opamp produces the output shown in figure 5.7.

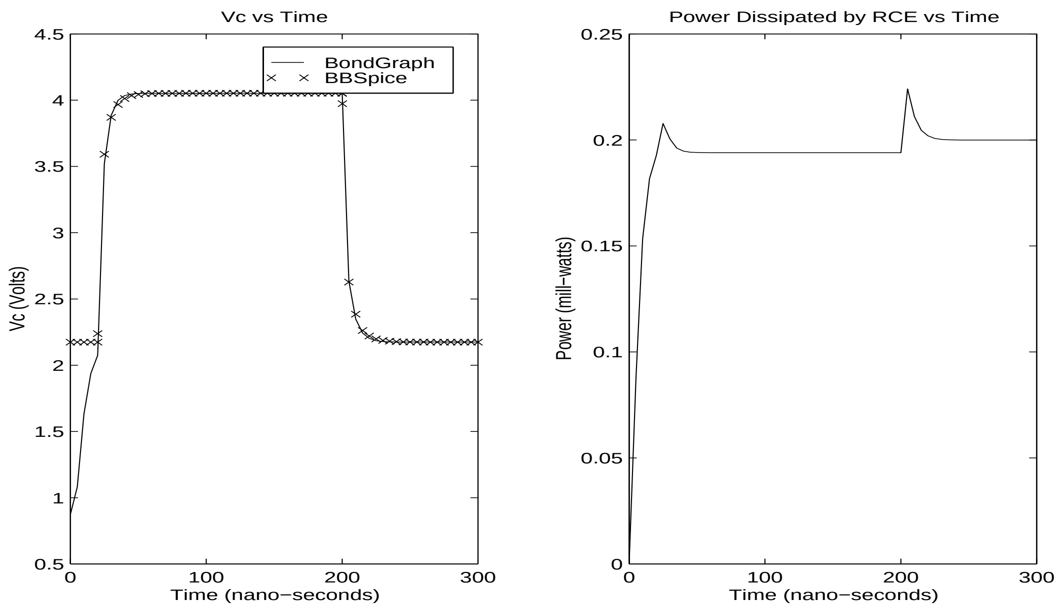

FIGURE 5.4. PNP Test Circuit Output (emitter voltage)

The Dymola model for the opamp produces the following number of equations:

Sequence of	367	equations
Sequence of	10	simultaneous equations
Sequence of	259	equations
Sequence of	14	simultaneous equations
Sequence of	730	equations
Sequence of	10	simultaneous equations
Sequence of	504	equations
Sequence of	10	simultaneous equations
Sequence of	684	equations

The opamp model takes approximately 19.4 seconds to simulate. This measurement does not include the time which is needed to compile the model.

5.5 The Second OpAmp Circuit

The opamp circuit found in section 5.4 is reused here to do a second experiment. In section 5.4, only one modulated capacitor was used to store the entropy flow produced by the twelve transistors found in the opamp model. Figure 5.8 shows the output

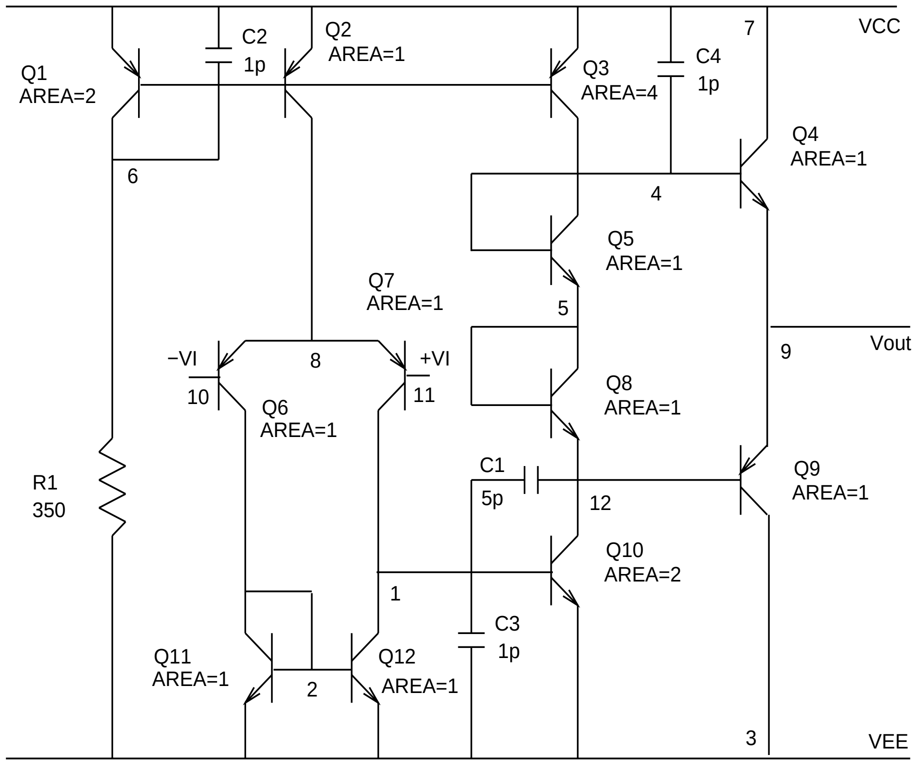

FIGURE 5.5. The Opamp Model Using Classical Standard Electronic Symbols

of this opamp circuit when the input voltage is fixed at two volts. When compared to the next plot (figure 5.9), the opamp's output falls slowly as a function of time. Figure 5.8 also shows the temperature of the opamp as a function of time.

Figure 5.9 shows what happens when each of the 12 transistors in the opamp is connected to a unique modulated capacitor. This configuration allows each of the transistors to have an independent temperature. The plot for this configuration predicts that the output voltage of the opamp rises when compared to figure 5.8.

The point of figures 5.8 and 5.9 is that the temperature has a feedback effect on the electrical circuit. This effect is currently not modeled by the BBSPICE or SPICE simulators. Typically, circuit designers try to put transistors as close together as possible so that they operate at the same temperature.

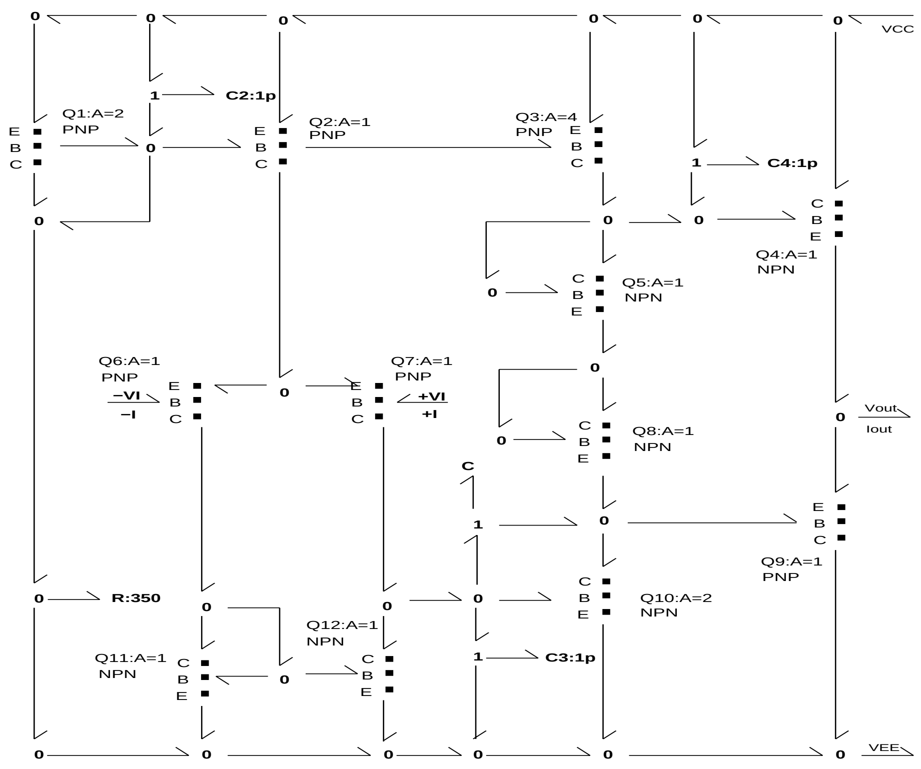

FIGURE 5.6. The Opamp Model Using Bond Graph Notation

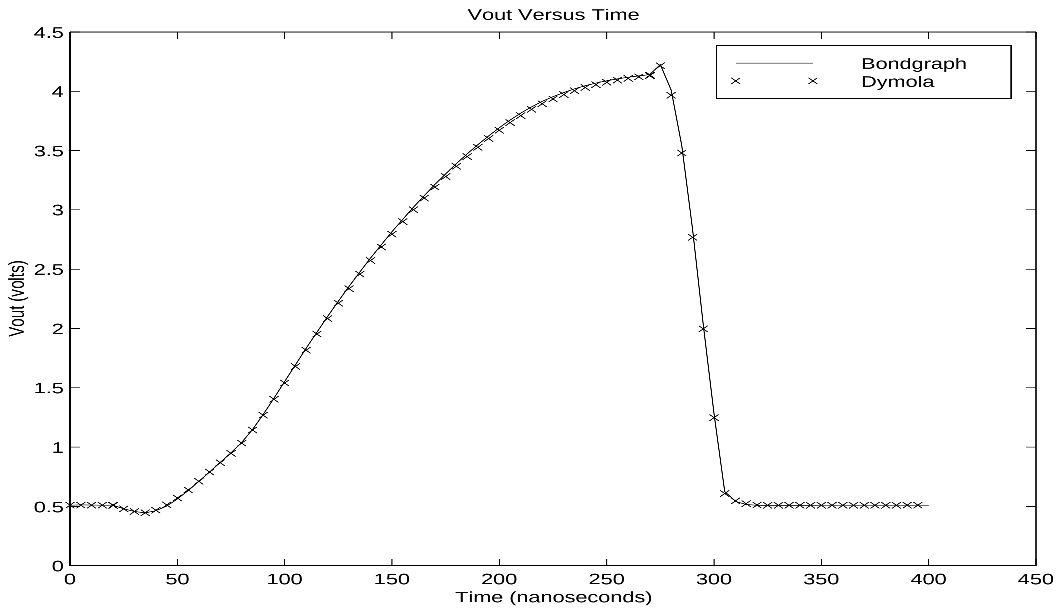

FIGURE 5.7. The Opamp Model Using Bond Graph Notation

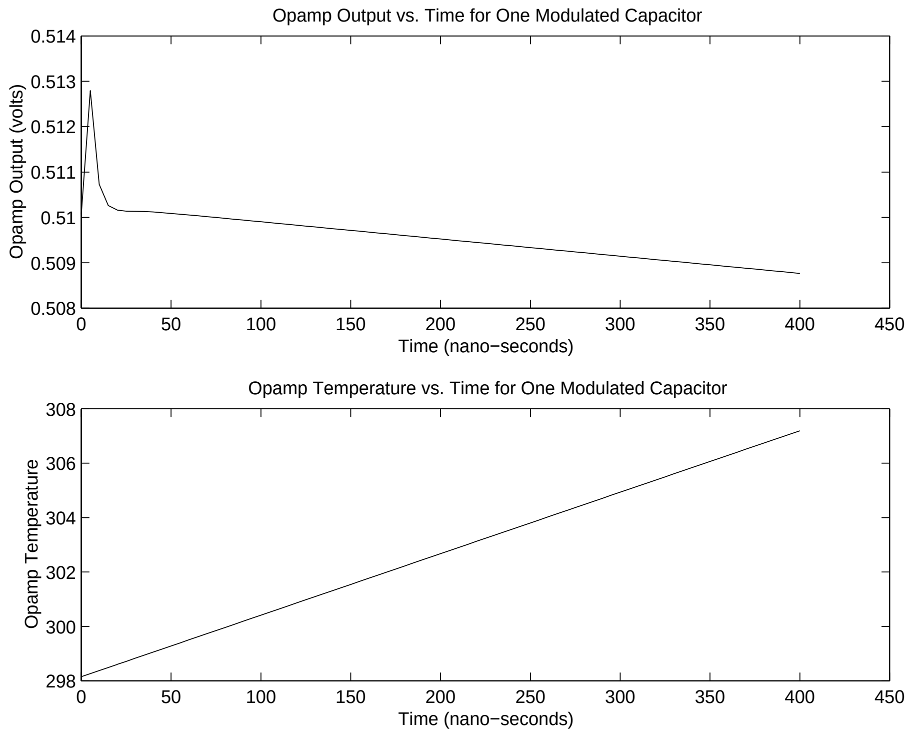

FIGURE 5.8. The Opamp Model Using Only One Modulated Capacitor

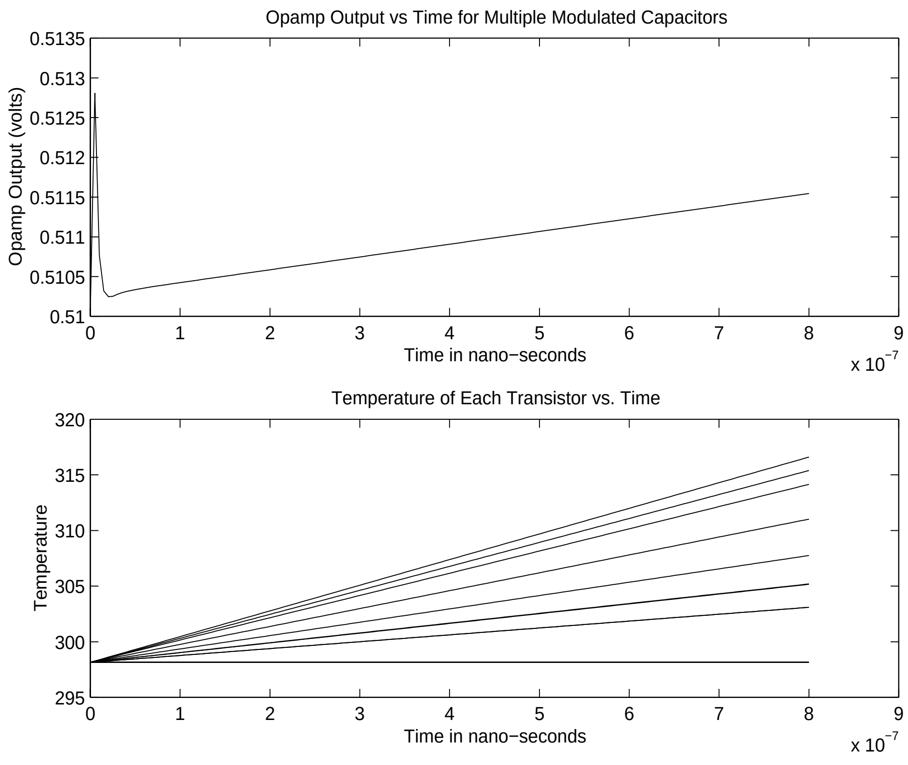

Figure 5.9. The Opamp Model Using Many Modulated Capacitors

Chapter 6 Conclusion

The goal of this thesis was to look at the analog circuit model of the BJT transistor from a bond graph perspective. The specific circuit model which was presented in this thesis was taken from Hild [7], Cellier [4], and Antognetti [1]. The bond graph perspective mandated that this analog circuit had to always dissipate power. This is because the energy in a closed system has to be constant and because the power that a circuit dissipates cannot simply disappear. Instead, it has to be transformed into thermal energy. The mathematical formulation for this concept was provided in chapter 3, and it is provided here again:

$$e * f = sdot * T \tag{6.1}$$

The simulation results of the bond graph BJT model were presented in chapter 5. These plots showed that the simulated BJT model did indeed always dissipate power since the *power* vs. *time* trajectories were positive semi-definite. Additionally, the plot of $Temperature$ vs. $Time$, as shown in chapter 4, is semi-monotonically increasing. Therefore, the plot shows that the power which is represented by $T * sdot$ is not transformed back into electrical power. Additionally, section 5.5 showed that the thermal performance of the model affects the electrical performance of the model via feedback.

In order to achieve these results, several things were developed in this thesis. The first was to correct and transform the models found in [4] and [7] into a model which was suitable for a bond graph. The Dymola modeling language was then used to create the components which were necessary to support the creation of bond graph models using the *Dymo Draw* modeling environment. Using this library, the bond graph model of a BJT was implemented and then simulated. Using the BJT models which were developed, an Opamp model was created and simulated to see how well Dymola could handle larger models.

The results of the simulations show that the transient responses which are predicted by [7] were duplicated, and in some cases, the small corrections that were made to the BJT model caused the resulting trajectories to match those which are predicted by BBSPICE.

Finally, because the BJT circuit model was shown to balance power, the models were shown to be physically justifiable.

While the BBSPICE simulator can perform all of the simulations presented in this thesis at many times the speed at which Dymola can currently do them, BBSPICE does not offer the flexibility, nor the extendibility that Dymola does. With a slight amount of overhead, the bond graph model was extended to not only model the electrical trajectories of the BJT, but also the thermal trajectories. Unlike BBSPICE, temperature was a variable of the Dymola model instead of a constant of the model. Hence, the thermal performance of the model could be fed back into the electrical side of the model. Finally, the physical soundness of the analog BJT model was verified by using bond graphs to show that the model does indeed conserve energy.

References

[1] Antognetti, Paolo, Massobrio, Giuseppe (1987), *Semiconductor Device Modeling with Spice* McGraw-Hill Book Company, New York.

[2] Blundell, Alan (1982), *Bond Graphs for Modelling Engineering Systems*, Ellis Horwood Limited, Chichester.

[3] Breedveld, P.C. (1984), *Physical Systems Theory In Terms of Bond Graphs*, Doctoral Thesis

[4] Cellier, F. (1991), *Continuous System Modeling*, Springer-Verlag New York Inc., New York, New York.

[5] Ghausi, Mohammed S. (1985), *Electronic Devices and Circuits: Discrete and Integrated*, Holt, Rinehart and Winston, New York.

[6] Greeneich, Edwin W. (1997), *Analog Integrated Circuits*, Chapman & Hall, New York.

[7] Hild, D.R. (1993), Circuit Modeling in Dymola, Dept. of Electr. & Comp. Engr., University of Arizona, Tucson, AZ.

[8] Homepage for Dymola, Dynasim AB Coorporation, http://www.dynasim.se.

[9] Muller, Richard S., Kamins, Theodore I. (1986), *Device Electronics for Integrated Circuits*, John Wiley & Sons, New York.

[10] Navon, David H. (1986), *Semiconductor Microdevices and Materials*, Holt, Rinehart, and Winston, New York.

[11] Neelakanta, Perambur S. (1995), *Electromagnetic Materials*, CRC Press, Boca Raton.

[12] The Bureau of Naval Personnel (1973), *Basic Electronics* Dover Publications New York, Inc., New York.

[13] Thoma, Jean U. (1990), *Simulation by Bondgraphs, Introduction to a Graphical Method*, Springer-Verlag, New York.

[14] Warner, R.M. Jr., Grung, B.L. (1983), *Transistors* John Wiley & Sons, New York.

[15] Whitaker, Jerry C., Editor-in-Chief (1996), *The Electronics Handbook*, Technical Press, Beaverton, Oregon.