fREEDA™

Programmer's Guide

Version 1.1.0

May 23, 2003

Contents

Chapter 1

fREEDA™ Architecture

1.1 Introduction

The architecture of fREEDA™ is based on faithful application of object oriented (OO) design practice [61–66] [1]. The OO abstraction is well suited to modeling engineering systems, for example, in circuit simulation circuit elements are already viewed as discrete objects and at the same time as an integral part of a (circuit) continuum. The OO view is a unifying concept that maps extremely well onto the way humans perceive the world around them. Non-OO circuit simulators always become complicated with many layers of special cases. Referring to circuit elements again, traditional simulation implementations have many "if-then" like statements and individually identify every element in many places for special handling.

There are a few key premises that drove the architecture of fREEDA™. One of these is the adoption of a very strong OO paradigm throughout to obtain a modular design. A second key premise is the separation of the core components embodying numerical methods from the modeling and solver formulation process with the result that numerical techniques developed by computer scientists and mathematicians can be formulated using formal correctness procedures. Thus, what is adopted here, is that the circuit abstraction is adapted so that highly reliable and efficient pre-developed libraries can be used.

C++ is the core implementation language and was once considered slow for scientific applications. Advances in compilers and programming techniques, however, have made this language attractive and in some benchmarks C++ outperforms Fortran [67, 68]. Several OO numerical libraries have been developed [69]. Of great importance to the work described here is the incorporation of the standard template library (STL) [70]. The STL is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. The current ISO/ANSI C++ standard [71] has not been fully implemented and C++ compilers support a variable subset of the standard. The biggest areas of noncompliance being the templates and the standard library.

In this chapter the focus is on the design of the object-oriented structure of fREEDA™. The goal in the design of fREEDA™ was to obtain speed in development, to use 'off-the-shelf' advanced numerical techniques, and to allow easy expansion and implementation of new models and numerical methods.

The design intent was to to combine the advantages of previous OO circuit simulators with these new developments as well as expanding capability. fREEDA™ uses C++ libraries [72, 73] and several written in C or Fortran [52, 74, 59].

OO-design embodies a body of concepts that should be understood before a full appreciation of what follows can be made. This fits in beautifully with the idea of code resue as here we utilize concept reuse. The first concept required is an understanding of the Unified Markup Language

[1] These references are available on line at http://www.objectmentor.com/

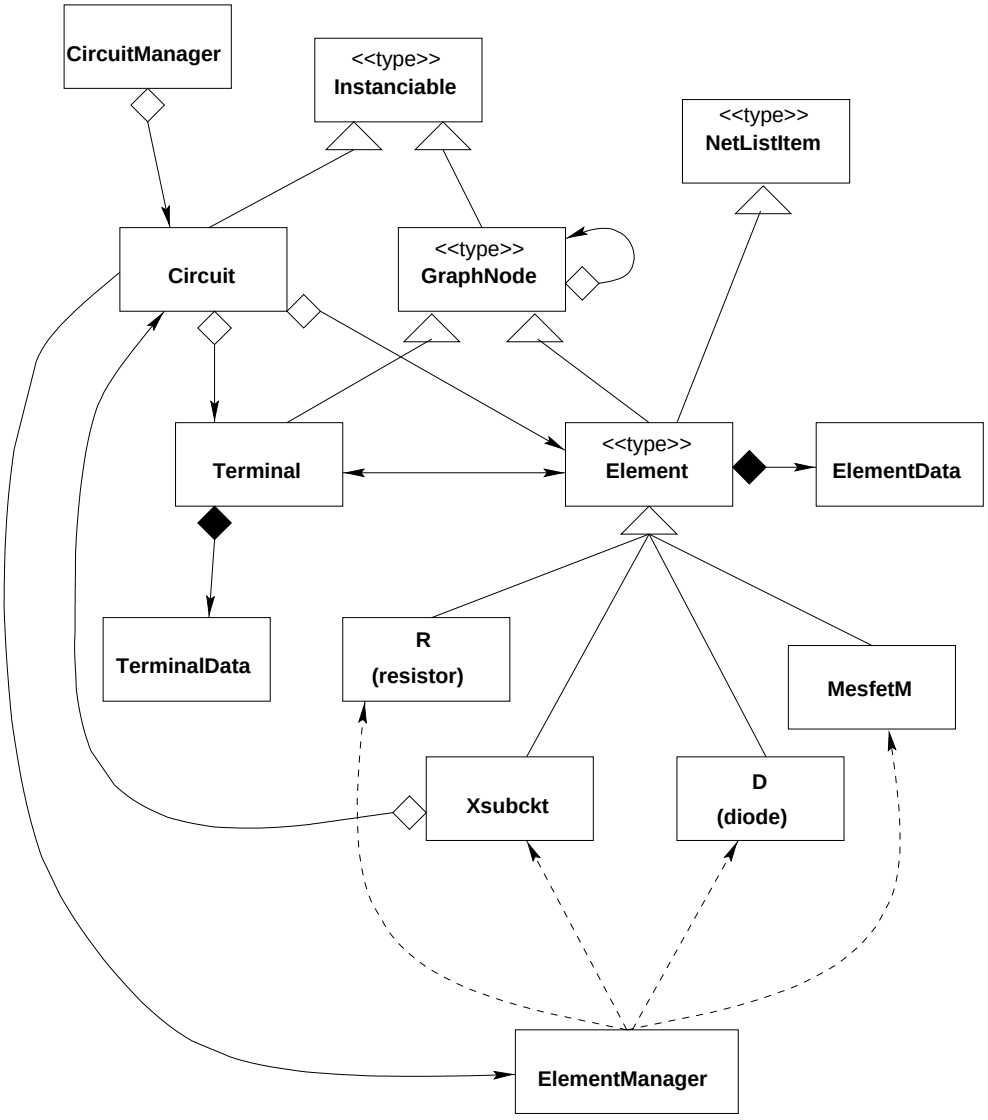

Figure 1.1: Class diagram: The network package is the core of the simulator.

(UML). This is a universal way of presenting relationships among objects and is not specific to computer coding. UML can be viewed as a substitute for flowcharts. A flowchart permits a very limited set of relationships to be described and results in what is called spaghetti code: code that has many linear streams and complex connections between the threads. Common code is multiplicated and perhaps slightly changed in each instance as the concept of using existing code and overriding only the differences is not supported. UML is different and fosters code reuse and much simpler coding. So if you are not familiar with UML study Appendix A before proceeding.

1.2 The Network Package

The network package is the core of the simulator. All the elements and the analysis classes are built upon it as shown in the class diagram of Figure 1.1. R(resistor), D(diode) and MesfetM shown here are examples of particular elements. Xsubckt is the `SPICE` X subcircuit element. The terminals of an element are specifically referred to as terminals and not as nodes as used in `SPICE` as the concept of a node is more general than that of a terminal. For example elements and terminals are both graph

Figure 1.2: The **NetListItem** class.

nodes. (See Appendix A for a description of the class diagram syntax.) Following the suggestion made in [9], there is a **NetListItem** class that is the base for all classes of objects that appear in the input netlist. This is the base class that handles parameters. Figure 1.2 shows some of the methods provided by this class. All the netlist items share a common syntax so that the element model developer does not need to worry about the details of element parsing and there is no need to modify the parser to add new elements. For compatibility reasons, Spice-type syntax (which does not have a consistent grammar) is supported by the parser outside the network package. Support for the elements using `SPICE` syntax is implemented for each element in the parser where each `SPICE` element is renamed as a *f*REEDA™ element.

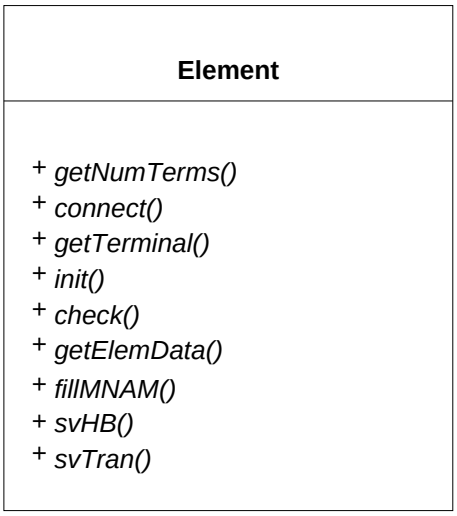

Figure 1.3: The **Element** class.

The **Element** class contains basic methods common to all elements as well as the interface methods for the evaluation routines. Some of the methods of this class (Figure 1.3) need to be overridden by the derived classes. For example, in class **D**, `svTran()` is intended to contain the code to evaluate the time domain response of a diode. This function is used by DC and transient analyses. The same happens with `svHB()` and `fillMNAM()`. The overhead imposed by these virtual functions is

small compared to the time spent evaluating the functions themselves and so this approach is a good compromise between flexibility and efficiency. This idea has been used in [8–10]. fREEDA™ also offers a more elaborate mechanism for nonlinear element evaluation functions which will be described in Section 1.4.

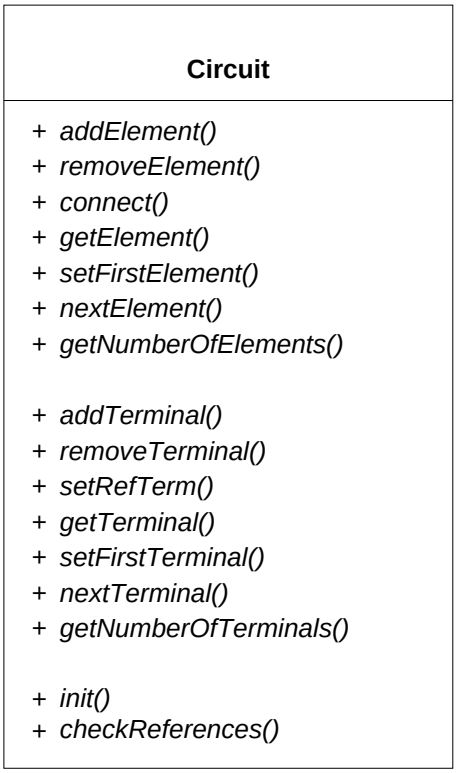

Figure 1.4: The **Circuit** class.

The **ElementManager** class is mainly responsible for keeping a catalog of all the existing elements. Note that this class is the only one that 'knows' about each and every type of element but this dependency is weak. The element list is included from an automatically generated file. **ElementManager** is also used to automatically generate documentation for the element in html format. The **Circuit** class represents either a main circuit or a subcircuit as a collection of elements and terminals. It provides methods to add, remove and find elements and terminals using different criteria and it also provides methods related to circuit topology. More details about this class are given in Figure 1.4. All the **Element** and **Terminal** instances must be stored in data structure inside the **Circuit** instance and the *map* container of the STL [70]. The map is a *Sorted Associative Container* that associates objects of type *Key* with objects of type *Data*. Here *Data* is either **Element** or **Terminal** and *Key* is *int* (the ID number). This is an example of where the features of C++ are used to reduce development time. This is achieved at no overhead as an optimum implementation of these concepts is embedded in the compiler. Subcircuit instances are represented by the **Xsubckt** class, see Figure 1.5. The method `attachDefinition()` is used to associate a **Circuit** instance where the actual subcircuit is stored to the particular **Xsubckt** instance and `expandToCircuit()` takes a **Circuit** pointer as argument and expands the subcircuit. Note that before expansion the complete hierarchy of a circuit is available in memory, so this engine could eventually be used to perform hierarchical simulation. The reason for this is that a subcircuit can be invoked (by an **X** element) before the subcircuit is defined (in a .SUBCKT block).

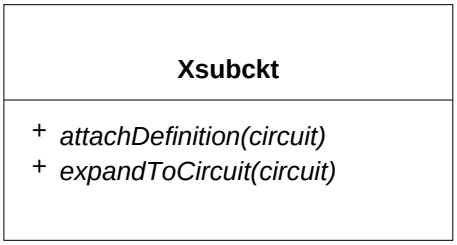

Figure 1.5: The **Xsubckt** class: This class handles the `SPICE` **X** element.

1.3 The Analysis Classes

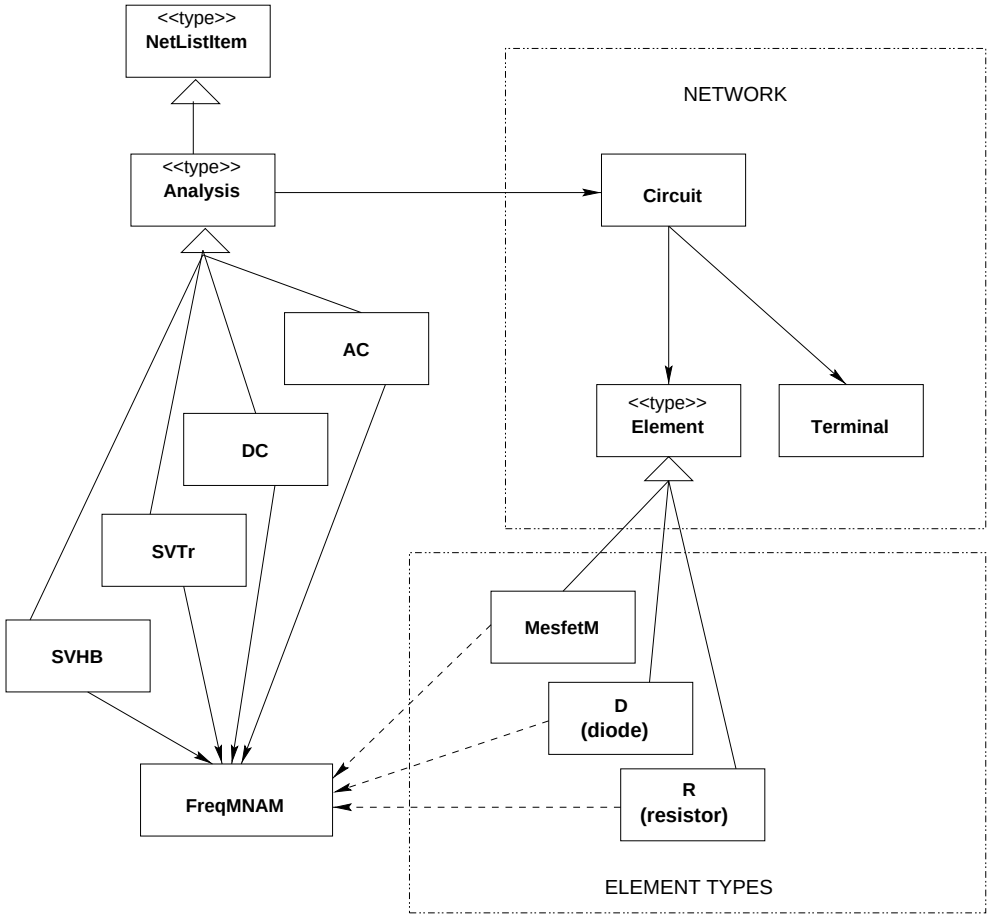

Figure 1.6: The analysis classes: `AC`, `DC`, `SVTr` and `SVHB` are different analysis types.

Figure 1.6 shows the relation between the network package, the elements and the analysis classes. Each of these classes stores analysis-specific data that would traditionally be global. This leads to the key desired attribute of flexibility in incorporating a new type of analysis, or even different implementations of the same analysis type. Examples are the **SVTr** and **SVHB** classes which contain the state-variable convolution transient and state-variable harmonic balance analyses described in [107].

There are some components common to two or more analysis types. The natural way of handling these components is by creating a class which is shared by the different analysis types. For example, the **FreqMNAM** class handles a Modified Nodal Admittance Matrix (MNAM) in the frequency

domain. In a microwave simulator, the frequency domain admittance matrix is a key element for most analysis types. Since it is used so often, special care was taken to optimize efficiency. The elements fill the matrix directly without the need for intermediate storage of the element stamp. They do that by means of in-line functions to reduce function call overhead. Elements can fill the source vector in a similar way. The elements depend on the **FreqMNAM** methods, but this is not a problem since the interface is very unlikely to change. The current implementation of MNAM uses the Sparse library, described in the next Section, and is completely encapsulated inside the **FreqMNAM** class. In the same way, the **NLSInterface** class encapsulates the nonlinear solver routines. Therefore it is possible to replace the underlying libraries, if that is desired, without the need for any code modification outside the wrapper classes. A final observation is that it is possible to add any kind of analysis type provided that the appropriate interface is defined and the member functions are written for each element type.

1.4 Nonlinear Elements

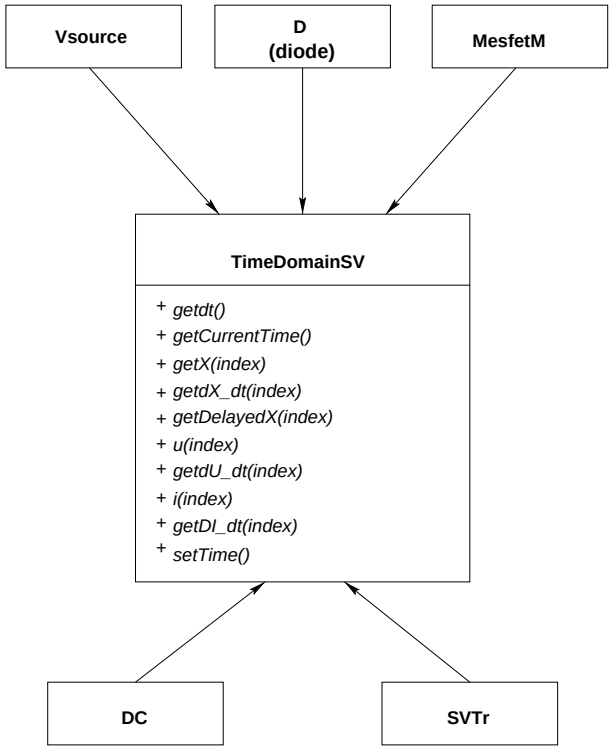

Figure 1.7: Dependency inversion was used to make the elements independent of the analysis classes.

Nonlinear elements often use service routines provided by the analysis classes. In order to maximize code reuse and to avoid the dependence of the element code on a particular analysis routines, in Transim elements depend on interface classes (Figure 1.7). The concept is similar to the *dependency inversion* [62]. This is a technique which relies on interface classes (normally implemented using abstract classes in C++) to make different parts of a program independent of each other. They only depend on the interfaces. To achieve greater efficiency, in Transim the dependency inversion is implemented using a concrete class with heavy in-lining and pass-by-reference.

In this way, the element routines and the analysis depend on an interface class, **TimeDomainSV** (not shown in Figure 1.1 for clarity). **TimeDomainSV** is a class that is used to exchange information between an element and a state variable based time domain analysis. It also provides some basic algorithms such as time differentiation methods. This approach enables the element routines

to be reused by several analysis types without the need to modify the element code (as long as the new analysis is state variable-based). For example, the **DC** analysis uses the same interface element as the **SVTr**.

*f*REEDA™ offers a more refined way to implement nonlinear elements, based on state variables (See chapter 3). By implementing those parametric equations using a special syntax in only one function, Transim can obtain the analysis functions `svHB()`, `svTran()` and derivatives automatically. This mechanism is termed *generic evaluation*.

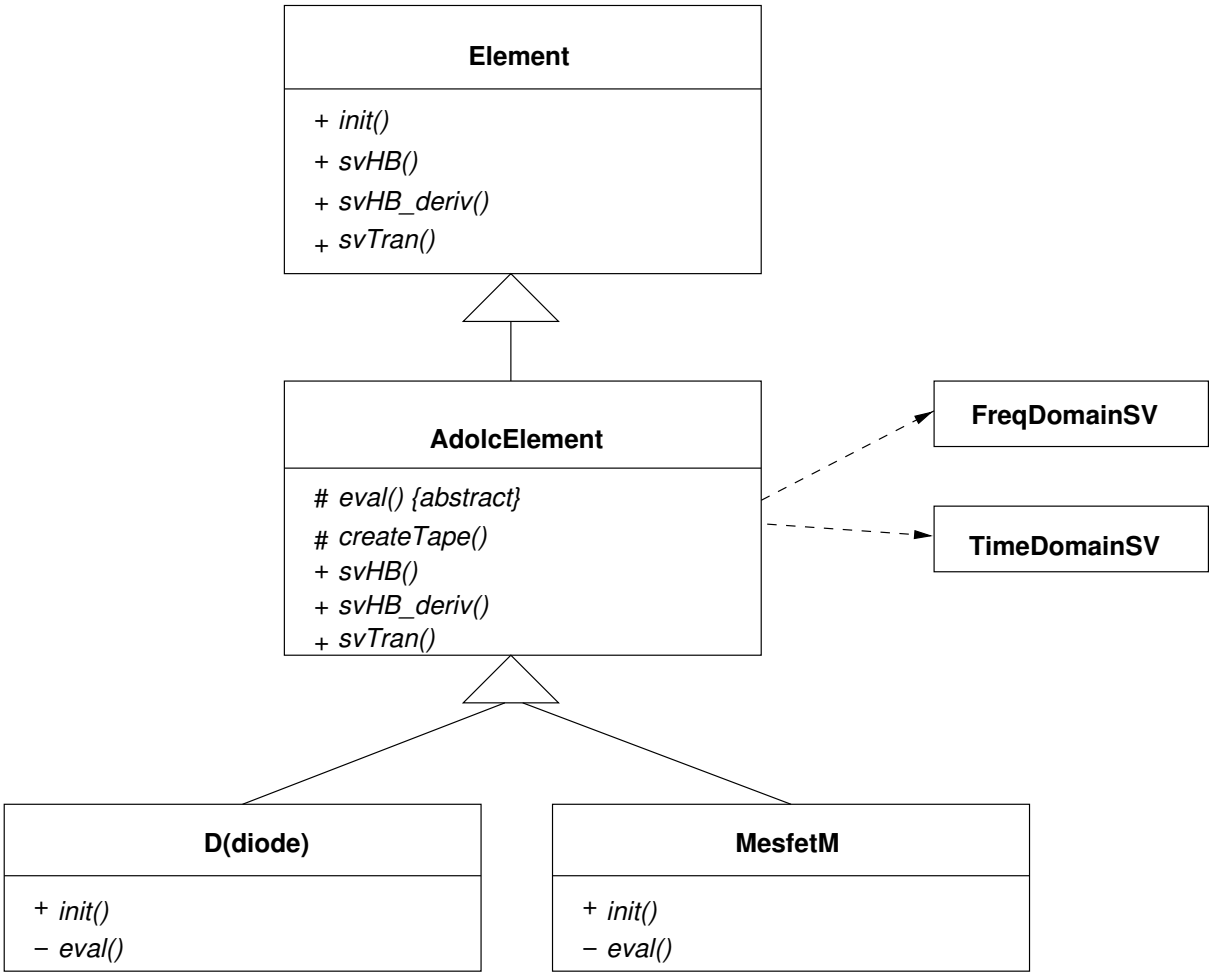

Figure 1.8: Class diagram for an element using generic evaluation. `D`(diode) and MesfetM are elements.

Figure 1.8 shows the class diagram for an element using this feature. Note that the **Diode** class is derived from a class (**AdolcElement**) which itself provides the analysis routines and deals with the analysis interfaces. The **Diode** class only needs to implement the `eval()` function with the parametric equations.

AdolcElement uses the Adol-C library (see Section C.5 for a detailed explanation) to evaluate the parametric function and derivatives, but the concept is independent of automatic differentiation. If automatic differentiation were not used, then the derived class (*e.g.* the **Diode** class) would have to provide the Jacobian for the parametric equations.

The idea is in a way similar to the one used in [7], *i.e.* the primitive equations are 'wrapped' in analysis-specific generic functions and so there is no need to write a separate routine for each analysis type. In the current work there are two additional features. The first is that the generic evaluation is combined with the state variable concept of Section **??** and automatic differentiation.

This provides unprecedented simplicity in creating nonlinear element models. The second is that a single mechanism is not mandatory. There are cases where it may not be practical to use this approach, or the overhead involved (which is completely acceptable even for simple models such as a diode) may not be properly amortized. In those cases, the element can implement the analysis functions directly, without using the **AdolcElement** class.

It is important to remark that generic evaluation is implemented efficiently so there are no superfluous calculations. The current implementation supports elements with any number of state variables. Each element selects the input variables as a subset of the following: the state variables, the first derivatives, the second derivatives and a time delayed version of the variables (the delay may be different for each). No derivation, time delaying nor transformation is performed on the unselected inputs.

For example, an element with only an algebraic nonlinearity (such as the VCT: voltage controlled transducer) only selects the state variables without any derivatives or delays. As another example, the **MesfetM** class implements the Materka-Kacprzac model for a MESFET. It requires two state variables, but only one of them needs to be delayed.

A consequence of having a library of elements using generic evaluation is that it is possible to add a new analysis type by just adding the appropriate evaluation routines to the **AdolcElement** class. Thus, the maintainance and expansion of the simulator is simplified. Put another way the code for a circuit element need only be implemented once and the identical code can be used by many different types of analyses including, hopefully any future analysis type implemented in the future.

1.5 Example: Use of Polymorphism

The concept of polymorphism is briefly explained in the introduction to object oriented programming in Appendix A. The scheme presented in Section 1.4 constitutes a good example of the use of polymorphism to solve a complex problem. Consider the segment of code of Figure 1.9. This code evaluates the nonlinear element functions in state variable convolution transient analysis. `elem_vec` is a vector of **Element** pointers implemented using the vector container of the C++ STL. There

```
// Number of elements
int n_elem = elem_vec.size();
int i = 0;
// Go through all the nonlinear elements
for (int k = 0; k < n_elem; k++) {
  // Set base index in interface object
  tdsv->setIBase(i);
  // nonlinear element evaluation
  elem_vec[k]->svTran(tdsv);
  i += elem_vec[k]->getNumberOfStates();
}
```

Figure 1.9: Nonlinear element function evaluation in convolution transient.

is no need to keep the size of the vector in a separate variable as `elem_vec.size()` returns the size of the vector. Also, the memory management of the vector is dynamic and automatic; and `elem_vec[k]` returns the Element pointer at position k in the vector.

Each pointer inside `elem_vec` points to different kinds of elements. For the transient routine the actual type of each element does not matter. The line containing `elem_vec[k]->svTran(tdsv)` will call the appropriate evaluation routine depending on the actual type of Element pointer. The element may implement the routine directly or through generic evaluation, but it makes no difference for the analysis routine.

The resulting code is therefore simple and there is no need for lists of “if-then” statements. Those lists would be very difficult to maintain, because each time an element is added or removed, all the lists would have to be updated.

1.6 Contributors

The following contributed to this chapter

Carlos Christoffersen

Michael Steer.

Chapter 2

Adding Linear Elements to fREEDA™

2.1 The fREEDA™ Circuit Simulator

*f*REEDA™ is a netlist-based circuit simulator. The input format of the netlist file is similar to the `SPICE` format with extensions for new device models, variables, sweeps, and repetitive simulation. The program provides a variety of output data and plots. The addition of new circuit element models and analysis types in *f*REEDA™ is much simpler than in other circuit simulators such as Spice. For example, new element models are coded and incorporated into the program without modification to the high-level simulator. The circuit analysis types currently available in *f*REEDA™ are DC, AC, harmonic balance (HB) [2], convolution transient [104], wavelet transient [105], and time-marching transient [107]. Some insight into the program architecture is given in [106].

This tutorial describes the addition of linear device models to *f*REEDA™[1]. We assume the reader is familiar with C++ syntax and basic concepts of object-oriented programming. Some issues such as the creation of an element with multiple reference nodes are not yet described.

In the majority of cases the code for a new element can be written by following the code for an existing similar element. For example, to write a new MOS transistor mode the code for an existing MOS model can be followed.

2.2 Example: Adding a Linear Resistor

We illustrate the addition of linear device models using a step-by-step example with a simple model: a linear resistor.

2.2.1 Netlist syntax

A brief description of the netlist will help in understanding the rest of this section.

The standard *f*REEDA™ netlist syntax is common to all elements. This differs from the `SPICE` syntax but this is also supported but is less general in that the grammar of the standard *f*REEDA™ netlist syntax is common to all elements while the grammar of the `SPICE` syntax is not consistent and each `SPICE` element must be handled separately in the parser.

[1] Further details may be found at http://www.freeda.org

The standard fREEDA™ netlist syntax is

⟨`netlist Name`⟩:⟨`element ID`⟩ ⟨`terminal1`⟩ ··· ⟨`terminaln`⟩ [⟨`parameter`⟩ = ⟨`value`⟩ ···]

The fREEDA™ netlist syntax for a resistor (the **r** element) is, for example, **r:r1 n1 n2 r=100.**

r:1 is the **elementID**, **n1** and **n2** are the names of two terminals and **r** is a parameter syntax name.

`SPICE` syntax is also supported for the original `SPICE` elements. In `SPICE` syntax the equivalent input is **r1 n1 n2 100**.

2.2.2 Class Name, Required Files, etc.

A good class name for the element is **Resistor**. By convention (in fREEDA™), class names should begin with capital letters and contain no underscores. We need a netlist name for our element. Assume we call it "r".

The files containing the model description are located in the `elements/` directory. There is a header file (`Resistor.h`) containing the declaration of the class that defines the element and a file containing the definition of the class with the actual model (`Resistor.cc`). Those are the only files needed to define this element.

2.2.3 The Header File

The header file starts with comment lines describing the element type and sometimes a simple ASCII drawing of the element schematic. The figure can be used to describe terminal numbering.

```
// This may look like C code, but it is really -*- C++ -*-
//
// This is an resistor model
//
//                res
//        o----/\/\/\----o
//
// by Carlos E. Christoffersen
```

Header files may be included more than once in C++ programs. To avoid multiple declarations of the classes defined in the body of the header file, the definitions in the header file are enclosed by following preprocessor directives:

```
#ifndef Resistor_h
#define Resistor_h
class Resistor : public
Element {
  ⋮
}
#endif
```

The **Resistor** class is derived from the base class **Element**. This is so for all elements in fREEDA™. For simple elements, the only public functions that must be declared are the following:

```
class Resistor : public Element
{
public:

  Resistor(const string& iname);
```

```
~Resistor() {}

static const char* getNetlistName()
{
  return einfo.name;
}

// fill MNAM
virtual void fillMNAM(FreqMNAM* mnam);

virtual void fillMNAM(TimeMNAM* mnam);
```

The constructor always takes the same argument, `iname`. This is the name of the particular instance of the resistor being created (*e.g.* "r1").

The destructor in this case is trivial. The next function, `getNetlistName()` must be defined in every *f*REEDA™ element. It returns the netlist name of the element ("r" in our example) and it is used during the netlist parsing. It is declared static because this function is called before the actual element instance is created.

The last two functions are the declarations of the functions to fill the entries corresponding to this element in the *modified nodal admittance matrix* (MNAM) of the circuit being simulated.

The private members of this class are:

```
private:

  // Parameter variables
  double r;

  // Element information
  static ItemInfo einfo;

  // Number of parameters of this element
  static const unsigned n_par;

  // Parameter information
  static ParmInfo pinfo[];

};
```

The netlist parameters of this element are declared as normal member variables. Here we have only one parameter called "r". There is no conflict with the netlist name of this element. The last three variables can be left unchanged for any other element in *f*REEDA™. We will describe their use later.

2.2.4 The Class Source File

The include preprocessor directives go first. The `ElementManager.h` file must be always included first. For linear elements, we also need to include the declarations for the MNAM classes. There are currently two implementations of the MNAM in *f*REEDA™. One in the frequency domain (`FreqMNAM`) and one in the time domain (`TimeMNAM`). The last included file is the declaration of the class for our element, `Resistor.h`.

```
#include "../network/ElementManager.h"
#include "../analysis/FreqMNAM.h"
#include "../analysis/TimeMNAM.h"
#include "Resistor.h"
```

Now we must define the static member variables:

```
// Static members
const unsigned Resistor::n_par = 1;

// Element information
```

```
ItemInfo Resistor::einfo = {
  "r",
  "Resistor",
  "Carlos E. Christoffersen",
  DEFAULT_ADDRESS"elements/Resistor.h.html"
};

// Parameter information
ParmInfo Resistor::pinfo[] = {
  {"r", "Resistance value (Ohms)", TR_DOUBLE, true}
};
```

"**r**" under Element information is the NetlistName. "**r**" under Parameter information is where the parameter name is defined.

The `n_par` variable must be equal to the number of parameters defined for the element. In our example, only one parameter is defined. Note the use of the scope (::) operator. It is used to indicate that the `n_par` variable is a member of the **Resistor** class. The `einfo` structure contains strings describing the element. The first string is the netlist name of the element ("r"). This name must always be given in lowercase letters. The second string is a more verbose description of the element. The third string should list the authors of the element code and the last string is used to store a web/ftp/e-mail address where more information about the element can be found. The idea here is give full exposure to the creator of an element and that an element is 'owned' by the person or group that creates the element.

The `pinfo` vector contains the description of each parameter defined in the element. The first field (always in lowercase) is the netlist name of the parameter followed by a more verbose description. The units of the parameter (if any) should be included in this description. The third field is a flag denoting the type of the parameter. The possible flags are defined in the `NetListItem.h` file. The corresponding types for each flag are given in Table 2.1. The last field in the parameter description is a flag that is *true* if the parameter

Flag	Corresponding type
TR_INT	`int`
TR_LONG	`long int`
TR_FLOAT	`float`
TR_DOUBLE	`double`
TR_CHAR	`char`
TR_STRING	`string`
TR_COMPLEX	`double_complex`
TR_BOOLEAN	`bool`
TR_INT_VECTOR	`IntVector`
TR_DOUBLE_VECTOR	`DoubleVector`
TR_INT_MATRIX	`IntMatrix`
TR_DOUBLE_MATRIX	`DoubleMatrix`

Table 2.1: Possible parameter types.

is required in the netlist (*i.e.* an error occurs if the parameter is not specified) and *false* otherwise.

The constructor definition follows:

```
Resistor::Resistor(const string& iname) : Element(&einfo, pinfo, n_par,
iname)
{
  // Value of r is required
  paramvalue[0] = &res;

  // Set the number of terminals
  setNumTerms(2);
```

```
  // Set flags
  setFlags(LINEAR | ONE_REF | TR_FREQ_DOMAIN);
}
```

The constructor for the **Element** class takes the static member variables we defined before and the instance name (`iname`) as arguments. In the body of the **Resistor** constructor we must point each element of the `paramvalue` vector to the address of the member variables corresponding to the netlist parameters. The order of the elements in the `paramvalue` vector must be the same order that we used in the description in `pinfo`.

The member function `setnumTerms()`, which is derived from the **Element** class, is used to set how many terminals our element has.

The last function used in this constructor sets some flags that are useful to classify the different elements in a circuit. Table 2.2 summarizes the meaning of each possible flag.

Flag	Meaning
LINEAR	Linear in HB and tran2 analyses.
NONLINEAR	Nonlinear in HB and tran2 analyses.
ONE_REF	One internal reference node (*e.g.* a normal element).
MULTI_REF	Two or more internal reference nodes.
TR_TIME_DOMAIN	The element is treated in time domain in convolution transient analysis. This includes nonlinear elements.
TR_FREQ_DOMAIN	The element is treated in frequency domain in convolution transient analysis. This includes linear elements.
SOURCE	The element is a source in tran2 analysis.

Table 2.2: Currently defined flags.

2.2.5 Filling the Modified Nodal Admittance Matrix

The last two methods in the source file are the routines used to fill the MNAM of the circuit. The first function takes a pointer to an object of the **FreqMNAM** class.

```
void Resistor::fillMNAM(FreqMNAM* mnam)
{
  // Ask my terminals the row numbers
  mnam->setAdmittance(getTerminal(0)->getRC(), getTerminal(1)->getRC(),
              one/res);
}
```

Here **setAdmittance** sets the four element stamp of an admittance **g**. Thus the stamp is

$$\begin{bmatrix} g & -g \\ -g & g \end{bmatrix} \tag{2.1}$$

The first terminal of the element is obtained using **getTerminal(0)** and the second terminal by **getTerminal(1)**. **getRC()** gets the row/column index and $g = 1/r$ (i.e. one/res) is the conductance of the element.

This is a class to represent a set of modified nodal admittance matrices in the frequency domain with their respective source vectors. The following is the list of functions available to fill the matrices and the source vectors:

```
//-------------------------------------------------------------------
// Methods to fill the matrices used by the elements
//-------------------------------------------------------------------
```

```
inline void setElement(const unsigned& row, const unsigned& col,
            const double_complex& val);


inline void setAdmittance(const unsigned& term1, const unsigned& term2,
              const double_complex& val);

inline void setQuad(const unsigned& row1, const unsigned& row2,
              const unsigned& col1, const unsigned& col2,
              const double_complex& val);

inline void setOnes(const unsigned& pos, const unsigned& neg,
              const unsigned& eqn);

//-------------------------------------------------------------------
// Methods to fill the source vectors
//-------------------------------------------------------------------

// Set one element of the current source vector
void setSource(const unsigned& row, const double_complex& val);

// Add value to a pair of elements of the current source vector
void addToSource(const unsigned& pos, const unsigned& neg,
          const double_complex& val);
```

The **FreqMNAM** class also provides a set of methods to retrieve information about the matrices. One of the most used in the element routines is:

```
// Get the current frequency
inline const double& getFreq();
```

This method returns the frequency at which the the MNAM is expected to be calculated.

The last method is used to fill the elements in the time-domain MNAM.

```
void Resistor::fillMNAM(TimeMNAM* mnam)
{
  // Ask my terminals the row numbers
  mnam->setMAdmittance(getTerminal(0)->getRC(), getTerminal(1)->getRC(),
              one/res);
}
```

For a simple element such as our resistor example, this function looks very similar to the frequency-domain case. Nevertheless, the interfaces of the **FreqMNAM** and **TimeMNAM** classes are not exactly the same. The following are the methods available to fill the MNAM in the time domain:

```
//-------------------------------------------------------------------
// Methods to fill the matrices used by the elements
//-------------------------------------------------------------------

// Methods for filling M
inline void setMElement(const unsigned& row, const unsigned& col,
              const double& val);

inline void setMAdmittance(const unsigned& term1, const unsigned& term2,
              const double& val);

inline void setMQuad(const unsigned& row1, const unsigned& row2,
              const unsigned& col1, const unsigned& col2,
              const double& val);

inline void setMOnes(const unsigned& pos, const unsigned& neg,
              const unsigned& eqn);
```

```
// Methods for filling Mp
inline void setMpElement(const unsigned& row, const unsigned& col,
             const double& val);

inline void setMpAdmittance(const unsigned& term1, const unsigned& term2,
              const double& val);

inline void setMpQuad(const unsigned& row1, const unsigned& row2,
          const unsigned& col1, const unsigned& col2,
          const double& val);

inline void setMpOnes(const unsigned& pos, const unsigned& neg,
          const unsigned& eqn);

//-------------------------------------------------------------------
// Methods to fill the source vector
//-------------------------------------------------------------------
// Reminder: reference source element is not stored

// Set one element of the source vector
inline void setSource(const unsigned& row, const double& val);

// Add value to a pair of elements of the source vector
inline void TimeMNAM::addToSource(const unsigned& pos, const unsigned&
neg, const double& val);

// Get the current time
inline const double& getTime();
```

Note that we must build two matrices denoted in the source code as `M` and `Mp`. These correspond to the **G** and **C** matrices in Chapter 3 of Reference [107]. Another difference with the frequency domain case is that the source vector is handled separately, as it will demonstrated in a future example.

2.2.6 Modifications to the rest of the fREEDA™ Source Files

Refer to the file `readme.txt` in the `freeda-0.1` subdirectory for instructions on compiling a new element into *f*REEDA™.

2.3 Template Files for New Linear Elements

The files in this section are provided to simplify the creation of new elements. All that is required is to replace `Element1` by the name of the actual class (or name of the element), copy the files with appropriate names in the `elements/` directory and write the code to fill the MNAM.

2.3.1 Header File

```
// This may look like C code, but it is really -*- C++ -*-
//
// This is a generic element template
//
//            +-----+
//        o----+ +----o
//            +-----+
//

#ifndef Element1_h
#define Element1_h 1
```

```
class Element1 : public Element

{

public:



  Element1(const string& iname);



  ~Element1() {}



  static const char* getNetlistName()

  {

    return einfo.name;

  }



  // fill MNAM

  virtual void fillMNAM(FreqMNAM* mnam);



  virtual void fillMNAM(TimeMNAM* mnam);
```

```
private:

  // Parameter variables (replace by your parameter variables here)
  double par1, par2;

  // Element information
  static ItemInfo einfo;

  // Number of parameters of this element
  static const unsigned n_par;

  // Parameter information
  static ParmInfo pinfo[];

};

#endif
```

2.3.2 Class Source File

```
#include ". ./network/ElementManager.h"
#include ". ./analysis/FreqMNAM.h"
#include ". ./analysis/TimeMNAM.h"
#include "Element1.h"

// Static members (set to the number of parameters of your element)
const unsigned Element1::n_par = ;

// Element information
ItemInfo Element1::einfo = {
  "res",
  "Element1",
  "Your Name",
  "http://your.web.page/element1.html"
};

// Parameter information
ParmInfo Element1::pinfo[] = {
  {"par1", "Parameter 1 (Ohms)", TR_DOUBLE, true},
  {"par2", "Parameter 2 (F)", TR_DOUBLE, false}
};
```

```
Element1::Element1(const string& iname) : Element(&einfo, pinfo, n_par,
iname)
{
  // Value of r is required
  paramvalue[0] = &par1;
  paramvalue[1] = &(par2 = 1e-12); // Example of how to set a default value.

  // Set the number of terminals (Set to the number of terminals of
  // your element)
  setNumTerms(2);

  // Set flags (do not change these for normal linear elements)
  setFlags(LINEAR | ONE_REF | TR_FREQ_DOMAIN);
}
void Element1::fillMNAM(FreqMNAM* mnam)
{
  // Put your model here
}
void Element1::fillMNAM(TimeMNAM* mnam)
{
  // Put your model here
}
```

2.4 Contributors

The following contributed to this chapter

Nikhil Kriplani

Carlos Christoffersen

Michael Steer.

Chapter 3

Adding Nonlinear Elements to fREEDA™

3.1 The fREEDA™ Circuit Simulator

This tutorial describes the addition of nonlinear device models to *f*REEDA™[1]. We assume the reader is familiar with C++ syntax and basic concepts about object-oriented programming.

Presently, there are two ways to write nonlinear element models, using two different base classes. One class, AdolcElement, has been extensively used and is the base class of the case study to follow. Unfortunately, a limitation recently discovered in the Adol-C library is reflected directly into a limitation on the elements that can be defined by the AdolcElement class. This class has three function prototypes for the `createTape()` function. One of these prototype functions is designed to allow the declaration of a second derivative state variable intended to track one of the declared state variables. `This second derivative capability has been found to be non-functional.` Usage of the function prototype is still permitted, as it is the only way to obtained delayed instances of state variables, but the argument for the second derivative IntVector to createTape() in this case should be 'novar' and this is noted herein.

The *f*REEDA™ team is engaged in the on-going development of a new nonlinear base class called NAdolcElement which is intended to alleviate the limitation in Adol-C and allow evaluation of second and third derivatives in models. While NAdolcElement is subject to change, model writers requiring these higher order derivatives are invited to inspect the header files and source codes for NMOSx and PMOSx elements (where x is 1,2,3, or 14) and the CompactDiode element and use them as templates for writing a model based on the NAdolcElement class. Model writers are encouraged to use NAdolcElement for writing all new elements, since over a period of time, AdolcElement will take on full legacy status and its continued use will be discouraged.

A future revision of this document will also contain a case study of an element based on the NAdolcElement base class.

3.2 Example: Adding a Nonlinear Electro-Thermal Resistor

We illustrate the addition of nonlinear device models using a step-by-step example with a simple model: a nonlinear electro-thermal resistor[2].

[1] Further details may be found at http://www.freeda.org

[2] Note that the electro-thermal resistor behaves as a linear resistor when the parameter "pdr" is set to *false*

3.2.1 Netlist syntax

A brief description of the ƒREEDA™ netlist syntax will assist in understanding of the details of model construction. The standard ƒREEDA™ netlist syntax is common to all elements. This differs from the `SPICE` syntax. `SPICE` syntax is also supported but its grammar is less general than the standard ƒREEDA™ netlist syntax. This is because the ƒREEDA™ grammar is common to all elements while the grammar of the `SPICE` syntax is not consistent and each `SPICE` element must be handled separately in the netlist parser.

The standard ƒREEDA™ netlist syntax is

⟨`netlist Name`⟩:⟨`element ID`⟩ ⟨`terminal1`⟩ ⋯ ⟨`terminaln`⟩ [⟨`parameter`⟩ = ⟨`value`⟩ ⋯]

For example, the ƒREEDA™ netlist syntax for the thermal resistor element (the **tr** element), which will be shown here for tutorial purposes, is:

tr:tr1 n1 n2 n3 n4 r0=100 l=1 w=1 t=35 rsh=100 pdr=true

tr1 is the **elementID** and **n1** through **n4** are the names of four terminals. **r0, l, w, t,** etc. are parameter syntax names, and the values on the right side of the = symbol are parameter values specific to this instance of **tr:1**. Some optional parameters have been omitted in this example.

3.2.2 Class Name, Required Files, etc.

A good class name for the element is **Tresistor**. By convention (in ƒREEDA™), class names should begin with capital letters and contain no underscores. A netlist name is needed for the element. Assume we call it "tr". *The netlist name must always be given in lowercase letters because the ƒREEDA™ netlist parser converts a netlist to lower case before attempting to identify elements or parameters.*

The files containing the model description are located in the `elements/` directory. There is a header file (`Tresistor.h`) containing the declaration of the class that defines the element and a file containing the definition of the class with the actual model (`Tresistor.cc`). Those are the only files needed to define this element. They are described next.

3.2.3 The Header File

The header file starts with comment lines describing the element type and sometimes a simple ASCII drawing of the element schematic as shown in Listing 3.1. The figure can be used to describe terminal numbering.

Listing 3.1: Header Comments of Tresistor.h

```
1
2       // This may look like C code, but it is really -*- C++ -*-
3       //
4       // This is an electro-thermal resistor model
5       //
6       //             ++++++++++++++
7       //             +      tr      +
8       //      t0  o------/\/\/\-------o  t1
9       //             +              +
10      //             +  *        *  +
11      //             +  |        |  +
12      //             ++ | ++++ | ++
13      //                |        |
14      //             t3  o       o  t2
15      //                tref    tout
16      //
17      // by Houssam S. Kanj
```

Header files may be included more than once in C++ programs. To avoid multiple declarations of the classes defined in the body of the header file, the definitions in the header file are enclosed by following preprocessor directives as in Listing 3.2.

Listing 3.2: Preprocessor Directives of Tresistor.h

```
17
18 #ifndef Tresistor_h
19 #define Tresistor_h 1
20
21 #include "../network/AdolcElement.h"
22
23
24 class Tresistor : public AdolcElement
```

The **Tresistor** class is derived from the base class **AdolcElement**. This is so for all nonlinear elements in *f*REEDA™ that rely upon ADOL-C for evaluating the first and higher derivatives of vector functions. For simple elements, the only public functions that must be declared are shown in Listing 3.3.

Listing 3.3: Class declaration and public functions of Tresistor.h

```
24
25 class Tresistor : public AdolcElement
26 { public:
27
28   Tresistor(const string& iname);
29
30   ~Tresistor() {}
31
32   static const char* getNetlistName()
33   {
34     return einfo.name;
35   }
36
37   // Do some local initialization
38   virtual void init() throw(string&);
39
40   // Get a vector with the indexes of the local reference nodes.
```

```
41   // Get terminal pointers in term_list
42   // ordered by local reference node:
43   //
44   // t0 t1 t2 t3
45   //
46   //    ^     ^
47   //  LRN1 LRN2
48   //
49   // local_ref_vec contains: {1, 3}
50   //
51   virtual void getLocalRefIdx(UnsignedVector& local_ref_vec,//
52                    TerminalVector& term_list);//
53
54   // fill MNAM
55   virtual void fillMNAM(FreqMNAM* mnam);
56
57   virtual void fillMNAM(TimeMNAM* mnam);
```

The constructor always takes the same argument, `iname`. This is the name of the particular instance of the thermal resistor being created (*e.g.* "tr1"). It is passed from the input netlist at the time fREEDA™ is run.

The destructor in this case is trivial. The next function, `getNetlistName()` must be defined in every fREEDA™ element. It returns the netlist name of the element ("tres" in our example) and it is used during the netlist parsing. It is declared static because this function is called before the actual element instance is created.

The function `init()` must be defined for every nonlinear element based on Adol-C, since it is here where we create the Adol-C tape. It is also here where we perform some initializations that cannot be done by the constructor, ***i.e.*** when some parameters are not known at construction time.

The function `getLocalRefIdx()` must be defined whenever we want to implement an element with multiple reference nodes ***e.g.*** spatially distributed elements or electro-thermal ones. It is important to know that this function has nothing to do with linearity. So we may have a linear element with one reference node, or with multiple reference nodes. We may also have a nonlinear element with one reference node, or with multiple reference nodes. Local reference nodes are covered in further detail in Section 3.2.4.

The last two functions are the declarations of the functions to fill the entries corresponding to this element in the *modified nodal admittance matrix* (MNAM) of the circuit being simulated. Please refer to [108] for a detailed explanation on how to implement a linear element.

The private members of this class are shown in Listing 3.4.

Listing 3.4: Private functions of Tresistor.h

```
58
59 private:
60
61
62   virtual void eval(adoublev& x, adoublev& vp, adoublev& ip);
63
64
65
66   // Element information
67   static ItemInfo einfo;
68
69   // Number of parameters of this element
70   static const unsigned n_par;
71
72   // Parameter variables
73   double r0, rsh, l, w, narrow, defw, t, tnom, tc1, tc2;
74   bool pdr;
75
76   // Parameter information
```

```
77   static ParmInfo pinfo[];
78
79   // The calculated resistor value
80      double r;
81 };
82
83 #endif
```

The netlist parameters of this element are declared as normal member variables. Here we have 11 parameters, and again we emphasize that *parameter names must always be given in lowercase letters because the fREEDA™ netlist parser converts a netlist to lower case before attempting to identify elements or parameters.* Please refer to reference [110] for a description of the parameters and the device equations. The variable "r" is a temporary variable used to compute the resistance value when the resistor is linear i.e. when pdr == *false*. The last three variables can be left unchanged for any other element in fREEDA™. We will describe their use later.

The only private function that must be defined is the `eval` function. It takes 3 arguments of type adoublev&, the first one is an input argument, and the second and the third are output arguments. Things will be more clear when we discuss the `createTape` function.

3.2.4 The Class Source File

Preprocessor Directives

The include preprocessor directives go first; see Listing 3.5. The `ElementManager.h` file must be always included first. The next file to be included is `AdolcElement.h`. This file must be included in all nonlinear elements that are based on Adol-C. Since our element is also a linear one, we also need to include the declarations for the MNAM classes. There are currently two implementations of the MNAM in fREEDA™. One is the frequency domain (`FreqMNAM`) and the other is the time domain (`TimeMNAM`). The last included file is the declaration of the class for our element, `Tresistor.h`.

Listing 3.5: Preprocessor directives of Tresistor.cc

```

2 #include "../network/ElementManager.h"
3 #include "../analysis/FreqMNAM.h"
4 #include "../analysis/TimeMNAM.h"
5 #include "Tresistor.h"
```

Static Variables and Model Parameters

Now we must define the static member variables as in Listing 3.6.

Listing 3.6: Static members and Parameter info of Tresistor.cc

```
7
8 // Static members
9 const unsigned Tresistor::n_par = 11;
10
11 // Element information
12 ItemInfo Tresistor::einfo = {
13   "tr",
14   "Tresistor",
15   "Houssam S. Kanj",
16   DEFAULT_ADDRESS"elements/Tresistor.h.html"
17 };
18
19 // Parameter information
20 ParmInfo Tresistor::pinfo[] = {
21   {"r0", "Resistance value (Ohms)", TR_DOUBLE, false},
```

```
22   {"l", "length (meters)", TR_DOUBLE, false},
23   {"w", "width (meters)", TR_DOUBLE, false},
24   {"t", "system Temperature (Celsius)", TR_DOUBLE, false},
25   {"rsh", "sheat resistance (Ohms/sq)", TR_DOUBLE, false},
26   {"defw", "default device width (meters)", TR_DOUBLE, false},
27   {"narrow", "narrowing due to side etching (meters)", TR_DOUBLE, false},
28   {"tnom", "initial Temperature (Celsius)", TR_DOUBLE, false},
29   {"tc1", "Temperature Coefficient (1/Celsius)", TR_DOUBLE, false},
30   {"tc2", "Temperature Coefficient (1/Celsius)", TR_DOUBLE, false},
31   {"pdr", "Power Depenent Resistor", TR_BOOLEAN, false}
32
33 };
```

The `n_par` variable must be equal to the number of parameters defined in our element. In our example, 11 parameters are defined. Note the use of the scope (::) operator. It is used to indicate that the `n_par` variable is a member of the **Tresistor** class. The `einfo` structure contains strings describing the element. The first string is the netlist name of the element ("tr"). *This name must always be given in lowercase letters because the fREEDA™ netlist parser converts a netlist to lower case before attempting to identify elements or parameters.* The second string is a more verbose description of the element. The third string should list the authors of the element code and the last string is used to store a web/ftp/e-mail address were to find more information about the element.

The `pinfo` vector contains the description of each parameter defined in the element. The first field (*again, always in lowercase*) is the netlist name of the parameter followed by a more verbose description. We recommend to include the units of the parameter (if any) in this description. The third field is a flag denoting the type of the parameter. The possible flags are defined in the `NetListItem.h` file. The corresponding types for each flag are given in Table 3.1. The last field in the parameter description is a flag

Flag	Corresponding type
TR_INT	`int`
TR_LONG	`long int`
TR_FLOAT	`float`
TR_DOUBLE	`double`
TR_CHAR	`char`
TR_STRING	`string`
TR_COMPLEX	`double_complex`
TR_BOOLEAN	`bool`
TR_INT_VECTOR	`IntVector`
TR_DOUBLE_VECTOR	`DoubleVector`
TR_INT_MATRIX	`IntMatrix`
TR_DOUBLE_MATRIX	`DoubleMatrix`

Table 3.1: Possible parameter types.

that is *true* if the parameter is required in the netlist (*i.e.* an error occurs if the parameter is not specified) and *false* otherwise.

Constructor for a Nonlinear Instance of Tresistor

The constructor definition for a nonlinear instance of `Tresistor` (where pdr == *true*) follows in Listing 3.7.

Listing 3.7: Constructor definition of Tresistor.cc

```
35
36 Tresistor::Tresistor(const string& iname) : AdolcElement(&einfo,
37 pinfo, n_par, iname) {
38
```

```
39   // Set parameters
40   paramvalue[0] = &(r0);
41   paramvalue[1] = &(l);
42   paramvalue[2] = &(w);
43   paramvalue[3] = &(t);
44   paramvalue[4] = &(rsh);
45   paramvalue[5] = &(defw = 1e-6);
46   paramvalue[6] = &(narrow = zero);
47   paramvalue[7] = &(tnom = 27);
48   paramvalue[8] = &(tc1= zero);
49   paramvalue[9] = &(tc2= zero);
50   paramvalue[10] = &(pdr=false);
51
52
53 }
```

The constructor for the **AdolcElement** class takes the static member variables we defined before and the instance name (`iname`) as arguments. In the body of the **Tresistor** constructor we must point each element of the `paramvalue` vector to the address of the member variables corresponding to the netlist parameters. The order of the elements in the `paramvalue` vector must be the same order that we used in the description in `pinfo`.

Then the `init()` function definition follows in Listing 3.8.

Listing 3.8: Initialization function of Tresistor.cc

```
54
55 void Tresistor::init() throw(string&)
56 {
57   if (!isSet(&w))
58     w = defw;
59   if (!isSet(&t))
60     t = tnom;
61
62   if (pdr){
63     // Set the number of terminals
64     setNumTerms(4);
65     // Set flags
66     setFlags(NONLINEAR | MULTI_REF | TR_TIME_DOMAIN);
67     // Set number of states
68     setNumberOfStates(2);
69
70     if (!isSet(&r0)) {
71       if (!isSet(&l))
72     throw("r0 or l must be specified for the resistance");
73       if (!isSet(&rsh))
74     throw("rsh must be specified for the resistance");
75     }
76     // create tape
77     IntVector var(2,0);
78     var[1] = 1;
79     createTape(var);
80   }
```

The `init()` function starts by doing some conditional assignments. The condition test cannot be carried out in the constructor since the parameters value are not known at construction time. Then if pdr == *true*, `i.e.` if it is an electro-thermal resistor, we use the member function `setnumTerms()`, which is derived from the **Element** class, to set how many terminals our element has.

The next function is used to sets some flags that are useful to classify the different elements in a circuit. Table 3.2 summarizes the meaning of each possible flag. Function `setNumberOfStates()` is used to set the number of state variables used to implement the model. It is important to know that the the functions `setnumTerms(), setFlags(), & setNumberOfStates()` could be called from the constructor, but in our

Flag	Meaning
LINEAR	Linear in HB and tran2 analyses.
NONLINEAR	Nonlinear in HB and tran2 analyses.
ONE_REF	One internal reference node (*e.g.* a normal element).
MULTI_REF	Two or more internal reference nodes.
TR_TIME_DOMAIN	The element is treated in time domain in convolution transient analysis. This includes nonlinear elements).
TR_FREQ_DOMAIN	The element is treated in frequency domain in convolution transient analysis. This includes linear elements).
SOURCE	The element is a source in tran2 analysis.

Table 3.2: Currently defined flags.

special case, we needed to put them in the `init()` function since we had to take some conditional decisions based on the parameters which are not known at construction time. Note, for example, that it is vitally important to set the NONLINEAR flag when an element requires the Adolc library facilities to implement a nonlinear function.

Next, the code makes some tests on the input parameters, and then throws an exception if some parameters are missing.

The final thing that you need to do is to create the Adolc Tape. This is done by calling the `createTape()` function with the proper arguments. All nonlinear elements that are derived from the `AdolcElement` class must do this.

First the statement *IntVector var(2,0);* create a vector called `var` of two elements and initialize them to 0. Then the statement *var[1] = 1;* initialize the second element to 1. Last, the statement `createTape(var)` creates the tape.

Building Arguments for `createTape( )`

We will digress from the discussion of the `Tresistor` model for a moment in order to discuss the process of creating state variables in some detail. The `createTape()` function is a member function of the `AdolcElement` class. There are three versions of it corresponding to three different valid function prototypes. The choice of function prototype depends upon the function that the model is to implement.

1: `createTape(IntVector& var)`

- Takes 1 argument, a vector of state variables.
- This is the form used in the **Tresistor** model.

2: `createTape(IntVector& var, IntVector& dvar)`

- Takes 2 arguments – a vector of state variables and a vector corresponding to 1st derivatives of state variables.
- This is the most widely used form, since many physical models use physical variables and their first derivatives.

3:
```
createTape(IntVector& var, IntVector& dvar, IntVector& d2var,
    IntVector& t_var, DoubleVector& delay)
```

- Takes 5 arguments – a vector of state variables, first derivatives, second derivatives, delayed state variables, and a vector of state variable delays.
- This form must be used 2nd derivatives or time delays are required in models.

For example, suppose that we need to create a new *f*REEDA™ element with the following description (**f**( ) can be **v**( ) or **i**( )):

$$\mathbf{f}(x_0, x_1, x_2, \frac{dx_0}{dt}, \frac{dx_1}{dt}, \frac{dx_2}{dt}, \frac{d^2x_2}{dt^2}, x_0(t-\tau_0), x_2(t-\tau_2))$$

Then the arguments of the `createTape( )` function would be as follows:

createTape(i_vec1, i_vec2, i_vec3, i_vec4, d_vec)

In this case 4 vectors of type `IntVector&` and one vector of type `DoubleVector&` are needed. The first one, `i_vec1`, consists of the indexes of the state variables. The second one, `i_vec2`, consists of the indexes of the derivatives of the state variables. The third one, `i_vec3`, consists of the indexes of the second derivatives of the state variables. The fourth one, `i_vec4`, consists of the indexes of the state variables that are delayed. The last vector, `d_vec`, contains the value of the delays applied to the state variables created by passing `i_vec4` to `createTape( )`. For this example, let τ_0 be 1 picosecond and τ_2 be 0.1 picosecond. To prepare this set of vectors for passing to `createTape( )`, the portion of C++ code in Listing 3.9 would appear in the `init( )` function prior to the call to `createTape( )`.

Listing 3.9: Example creating input vectors passed to createTape( )

```
1  // Set number of states
2  setNumberOfStates(3);    // Three main state vars x0, x1, and x2
3
4  // Argument vector for creating main state variables
5  IntVector i_vec1(3,0);  // Initialize 3-element integer array to 0
6  i_vec1(1) = 1;          // Set elements for additional state variables
7  i_vec1(2) = 2;
8
9  // Argument vector for 1st derivatives of state variables
10 IntVector i_vec2(3,0);  // Initialize 3-element integer array to 0
11 i_vec2(1) = 1;          // Set elements for additional 1st derivatives
12 i_vec2(2) = 2;
13
14 // Argument vector for 2nd derivatives of state variables
15 IntVector i_vec3(1,0);  // Initialize 1-element integer array to 0
16 i_vec3(0) = 2;          // Set element to get 2nd derivative of state var x2!
17
18 // Argument vector for delayed state variables
19 IntVector i_vec4(2,0);  // Initialize 2-element integer array to 0
20 i_vec4(1) = 2;          // Set elememement to get delayed copy of state var x2!
21
22 // Argument vector for time delays
23 DoubleVector d_vec(2,0);     // Init. 2-element double array to 0
24 d_vec(0) = 1e-12;            // Delay state variable x0 by 1 picosecond
25 d_vec(1) = 1e-13;            // Delay state variable x2 by 0.1 picosecond
```

Notice that `i_vec1` and `i_vec2` are equivalent, so in this case it was not necessary to create `i_vec2`, and one could instead have simply passed `i_vec1` to `createTape( )` twice, in succession:

createTape(i_vec1, i_vec1, i_vec3, i_vec4, d_vec)

This emphasizes the point that the variables of type `IntVector&` and `DoubleVector&` are used only to create the collection of state variables and their derivatives and delayed versions. A survey of the source code library for existing *f*REEDA™ elements will reveal many instances of this sort of variable re-use in calls to `createTape( )`.

After the completion of the call to `createTape( )`, a vector variable `x[]` of type `adoublev&` will have been created. `x[]` is limited in scope to the instance of the model object. The number of elements in `x[]` will be equal to the number of unique mathematical objects requested in the call to `createTape( )`. For the

example here, there are 3 state variables, 3 first derivatives of state variables, 1 second derivative of a state variable, and 2 delayed state variables, so the total number of elements in `x[]` will be 9, with indices running from 0 to 8. State variables and their derivatives or delays are assigned indices in `x[]` consistent with the order of argument passage to `createTape( )`. Table 3.3 summarizes the relationship between vector input variables to `createTape( )` and the resulting state variables created by `createTape( )` for the example call described here:

Table 3.3: State Variable representation in *f*REEDA™.

Physical to State Variable Mapping	
Physical Variable	X Variable
x_0	x[0]
x_1	x[1]
x_2	x[2]
dx_0/dt	x[3]
dx_1/dt	x[4]
dx_2/dt	x[5]
d^2x_2/dt^2	x[6]
$x_0(t-\tau_0)$	x[7]
$x_2(t-\tau_2)$	x[8]

One final note on constructing the arguments to `createTape( )`: There are times when the implementation of a complex function may not require the creation of state variables corresponding to each argument in the most complex function prototype for `createTape( )` . For example, suppose in the previous example, that we did not need the second derivative of the second state variable, but we still needed the time-delayed variables. Then instead of declaring an IntVector called `ivec_3` of non-zero length, we would declare vector without a declared length, or a null vector, and pass this as an argument to `createTape( )` instead of `ivec_3`. It is common in many *f*REEDA™elements for this null vector to be called `novar`. For example, if indeed we wished to ignore second derivatives in the example in Listing 3.9, then we would alter lines 14 through 16 as follows in Listing 3.10 and then call `createTape( )` as follows:

createTape(i_vec1, i_vec2, novar, i_vec4, d_vec)

Listing 3.10: Declaring a null vector argument for createTape( )

```
14 // Argument vector for 2nd derivatives of state variables
15 IntVector novar          // Initialize a null vector (no elements)
16                          // Nothing to set!
```

Notice: A limitation in the functionality of the Adol-C Library has recently been uncovered: The second derivative of state variables, normally invoked by using the most complex form of the createTape prototype, has been determined to be non-functional. Thus, the use of novar as noted in listing 3.10 is mandatory. The development of the NAdolcElement class is an attempt to work around this limitation. See section 3.4 for further information.

Returning our attention to the `Tresistor` element, the function being implemented was `f(`x_0 `,` x_1`)`, so that only one vector filled with 0, and 1 was needed. To better understand the process of creating the Adolc Tape, model writers are advised to read the implementation of the `createTape()` function which is a member function of the `AdolcElement` Class. Model writers are also advised to read the ADOL-C documentation [109].

Constructor for Linear instance of Tresistor

The rest of the code in the `init()` function implements the linear part of the Constructor (invoked when pdr == *false* in the netlist instantiation) and is shown in Listing 3.11. Please refer to [108] for more details.

Listing 3.11: Init function of Tresistor.cc (continued)

```
16
17   else{  // pdr==false
18     // Set the number of terminals
19     setNumTerms(2);
20     // Set flags
21     setFlags(LINEAR | ONE_REF | TR_FREQ_DOMAIN);
22     // calculate the resistor value
23     if (isSet(&r0)){
24       r = r0*(1+tc1*(t-tnom)+tc2*(t-tnom)*(t-tnom));
25     }
26     else {
27       if (!isSet(&l))
28     throw("r0 or l must be specified for the resistance");
29       if (!isSet(&rsh))
30     throw("rsh must be specified for the resistance");
31       r = rsh*((l-narrow)/(w-narrow))*(1+tc1*(t-tnom)+tc2*(t-tnom)*(t-tnom));
32     }
33   }
34 }
```

Local Reference Nodes and the getLocalRefIdx( ) function

We will digress again from the `Tresistor` model for a moment to discuss the local reference node concept and some related modelling conventions inherent in *f*REEDA™.

A spatially distributed circuit is a circuit which, by definition, is much larger than the wavelengths of the signals it processes. In such circuits, the notion of a single reference node or ground terminal is inapt, since currents passing through such a reference node from one piece of spatially distributed circuitry to another cannot physically do so instantaneously. To overcome this limitation and permit modelling of circuits which may be spatially distributed, the local reference node concept was developed. Put adroitly (but in a way that should be widely understood), local reference nodes allow the definition of multiple "ground" terminals so that spatially distinct circuitry can reference their local circuits to these physically isolated reference nodes. Readers interested in understanding more about the theory and the implementation of the local reference node concept should refer to [111] and [112].

The availability of local reference nodes in *f*REEDA™ enables the definition of spatially distributed circuit models. Before discussing the coding details, which are straightforward, some modelling conventions used in *f*REEDA™ will first be described. These conventions are familiar to researchers in the circuit simulation community, but may not be familiar to new *f*REEDA™ users who wish to write *f*REEDA™ models.

Consider the "Blob" circuit shown in Figure 3.1. Blob is a circuit that has 5 terminals and is spatially distributed. It has three circuit terminals with terminal numbers 0, 1, and 3, and two reference nodes which have terminal numbers 2 and 4, as shown in the figure. Circuit terminals 0 and 1 use terminal 2 as a local reference node and circuit terminal 3 uses terminal 4 as a local reference node. Although details of the internal operation of "Blob" may not be known, it is clear that the circuits associated with terminals 0 and 1 are spatially distinct from the circuits associated with terminal 3.

Note that in all *f*REEDA™ model *source code definitions*, terminal number 0 is *always* a circuit terminal. This may seem incorrect to SPICE users familiar with using node 0 as the ground terminal in netlist definitions. However, note that a *f*REEDA™ netlist still permits the use of node 0 in the netlist for the ground terminal. Bear in mind the distinction between coding a model definition, which we are doing here, and instantiating a model, which is done in a netlist.

Figure 3.1 contains other information worth noting. First, note that each state variable is assigned, in order, to each *non-reference node* in the model. Second, notice the mention of the voltage variables `vp[]`

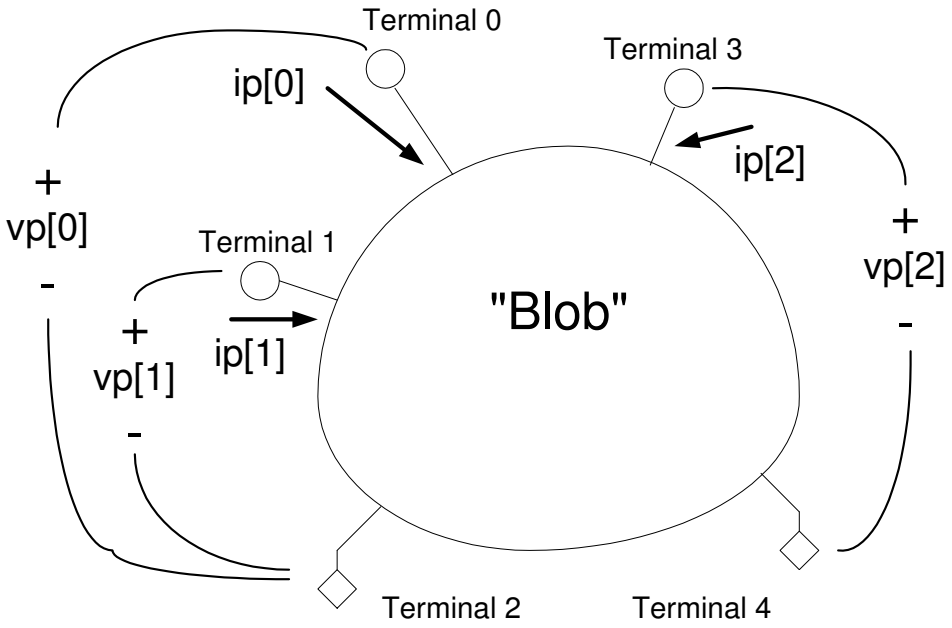

Figure 3.1: Spatially-distributed Blob circuit.

and the current variables `ip[]`. These variables are also initially created in the call to `createTape( )`, and are made available to each model instance's `eval()` function. (The `eval()` function will be discussed in detail in section 3.2.4 .) For each state variable created by the first argument in the call to `createTape()`, a corresponding `vp[]` and `ip[]` variable is created. Third, note that by convention, the direction of the current variable `ip[]` is into the terminal, and the voltage `vp[]` is defined positively at the circuit terminal and negatively at the corresponding reference node. Users wishing to define currents or voltages contrary to this convention may do so by simply negating the numerical expressions for `vp[]` and `ip[]` that appear in the model's `eval()` function. Finally, note that there are no state variables associated with the reference nodes, since voltages are defined relative to them and Kirchoff's current law requires that the sum of the non-reference terminal currents associated with a particular reference node must flow into that reference node.

Those accustomed to having a "port" perspective for models will notice that, in general, the number of ports is equal to the number of required state variables, even if local reference terminals are shared among one or more ports.

Returning now to the model code, implementation of the `getLocalRefIdx()` function is very simple and straight forward. Refer to Listing 3.12. The first two lines must always be included. Then, we push back each terminal in the terminal list by writing `term_list.push_back(getTerminal(n));` where "n" is the index of the terminal. When we push back a local reference node we must also push the node back into the local reference vector by writing `local_ref_vec.push_back(m);` where "m" is the index of the terminal that we want it to be a local reference.

The `getLocalRefIdx()` function definition follows in Listing 3.12.

Listing 3.12: getLocalRefIdx function of Tresistor.cc

```
122
123 void Tresistor::getLocalRefIdx(UnsignedVector& local_ref_vec,
124                 TerminalVector& term_list)
125 {
126   // Make sure the vectors are empty
127   term_list.erase(term_list.begin(), term_list.end());
```

```
128   local_ref_vec.erase(local_ref_vec.begin(), local_ref_vec.end());
129
130   // Insert vector elements
131   term_list.push_back(getTerminal(0));
132   term_list.push_back(getTerminal(1)); // Local reference terminal
133   local_ref_vec.push_back(1); // Local reference index
134
135   if (pdr){
136     term_list.push_back(getTerminal(2));
137     term_list.push_back(getTerminal(3)); // Local reference terminal
138     local_ref_vec.push_back(3); // Local reference index
139   }
140
141 }
```

Finally, note that if a model has only 1 local reference node, then it is not necessary to declare the getLocalRefIdx function at all in the header file and it may be omitted from the source code file. Note that when getLocalRefIdx is not used, then the highest numbered terminal is defined as the local reference terminal. Referring to listings 3.8 and 3.11 where the call to `setNumTerms()` appears, if `n` is the argument to `setNumTerms()`, then there will be `n` terminals numbered `0,1,..,n-1`, and when getLocalRefIdx is not used, then `n-1` will be the terminal number of the reference node.

The Eval function and using `condassign()`

The `eval()` function definition for `Tresistor` is shown in Listing 3.13.

Listing 3.13: Eval function of Tresistor.cc

```
100
101 void Tresistor::eval(adoublev& x,
102              adoublev& vp, adoublev& ip)
103 {
104   // x[0]: resistor voltage
105   // x[1]: deltatemp in deg. Celsius
106
107   vp[0]=x[0];
108   vp[1]=x[1]+tnom+273; //vp[1]==tp[1] in Kelvin
109
110   adouble res;
111   if (isSet(&r0)){
112     res = r0 * (one + tc1 * x[1] + tc2 * x[1] * x[1]);
113   }
114   else {
115     res = rsh * ((l-narrow) / (w-narrow))
116       * (one + tc1*x[1] + tc2 * x[1] * x[1]);
117   }
118
119   ip[0] = x[0] / res;
120   ip[1] = - x[0] * ip[0]; //ip[1]==pp[0];
121 }
```

The `eval()` function is the implementation of the device equations. It is in here where the voltages and currents are related to the state variables. The `eval()` function takes three vectors of type `adoublev&`, the first one is the state variable vector, and it is considered as the input, the second and the third one are the voltages and the currents, and these two vectors are considered as output. The `eval()` function is called from the `createTape()` function. In this way, the Adol-C library keeps track of all operations that are performed on the `adouble` variables so that it can evaluate the derivatives whenever needed.

Due to encapsulation of most of the low-level handling of the state variables within AdolcElement, manipulation of the state variables within the `eval()` function for an element can be done in a straightforward

fashion using the same operators as that for floating-point arithmetic. These operators are all overloaded, but this is transparent in most cases. However, when conditional branches affecting the assignment of active variables are to be made within an `eval()` function, it is mandatory to use a routine with the Adol-C library designed to facilitate conditional branching. This routine is called `condassign( )`.

The `condassign( )` routine normally takes 4 arguments which can be of type adouble or scalars. For users familiar with C and C++, a function call such as `condassign(a,b,c,d)` corresponds with the syntax of the following conditional assignment:

$$\mathbf{a = (b > 0)?\ c : d}$$

For those familiar with Matlab, Listing 3.14 shows how the same conditional branching statements might be implemented in that language.

Listing 3.14: Matlab equivalent of `condassign()`

```
1 if (b > 0)
2     a = c
3 else
4     a = d
5 end
```

As an illustrative example, consider a rudimentary model for a semiconductor junction diode. Define the current as being 0 whenever the voltage across the junction is less than, say, 0.7 volts, and let the current be non-zero above the threshold. Let voltage be the parametric state variable, so $v(t) = x(t)$, and let the current be defined as follows:

$$i(t) = \begin{cases} I_s\,[exp(\frac{x(t)-0.7}{v_T}) - 1] & \text{if } x(t) > 0.7 \text{ or } x(t) - 0.7 > 0 \\ 0 & \text{if } x(t) \leq 0.7 \end{cases}$$

In this diode equation, I_s is the reverse saturation current and v_T is the "temperature voltage." Both are constants. Letting `x[0]` be the state variable parameter $x(t)$, `vp[0]` be the junction voltage $v(t)$, and `ip[0]` be the current $i(t)$ across the junction, Listing 3.15 shows how the conditional branch for the simple junction diode model would be implemented.

Listing 3.15: Example of branch for simple junction diode.

```
1 vp[0] = x[0];                              \\ voltage equals parameter
2 condassign(ip[0], x[0]-0.7,                \\ a, b with b's condition shifted to 0.7
3              Is*(exp((x[0]-0.7)/vt)-1),    \\ c == exponential characteristic
4              0);                           \\ d == 0
```

The fillMNAM function

The last two methods in the source file, in Listing 3.16, are the routines used to fill the MNAM of the circuit. Please refer to [108] for a detailed explanation on how to fill the MNAM.

Listing 3.16: fillMNAM function of Tresistor.cc

```
144
145 void Tresistor::fillMNAM(FreqMNAM* mnam)
146 {
147   // Ask my terminals the row numbers
148   mnam->setAdmittance(getTerminal(0)->getRC(), getTerminal(1)->getRC(),
149                       one/r);
150 }
151
152
153 void Tresistor::fillMNAM(TimeMNAM* mnam)
154 {
```

```
155   // Ask my terminals the row numbers
156   mnam->setMAdmittance(getTerminal(0)->getRC(), getTerminal(1)->getRC(),
157                 one/r);
158 }
```

3.2.5 Using the cout routine for debugging

In the course of developing models, debugging will inevitably be necessary. In this section we provide a few suggestions on using the C++ standard output routine `cout` to facilitate debugging. Often in model development, modelling equations are valid over restricted portions of the input domain. Considering the rudimentary p-n junction diode again, suppose the current is undefined for voltages below the threshold voltage, rather than set to zero. We dispense entirely with the `condassign()` statement, but now it would be useful to have some sort of error notification to the user if voltages below the threshold are present. Listing 3.17 shows an example of how the code for the rudimentary diode might be modified to use `cout` to print such a message.

Listing 3.17: Using `cout` to facilitate model debugging.

```
1 vp[0] = x[0];                                  \\ voltage equals parameter
2 if (x[0] < 0.7)
3     {
4     cout << "x =" << x[0] << ".. Invalid input voltage." << endl;
5     }
6 ip[0] = Is*(exp((x[0]-0.7)/vt)-1);             \\ exponential characteristic
```

Another use for `cout` is to validate that model parameters are being passed as expected from the netlist into model instances. In these cases, `cout` statements would appear in the `init()` function call for the element and echo the parameter values of the model instance.

Note that when an active variable is streamed to the standard output, the real value of the variable is followed by an "(a)" to indicate that the variable is active. This is a result of the overloading of the "<<" operator by the Adol-C library.

3.2.6 Modifications to the rest of the fREEDA™ Source Files

Refer to the file `readme.txt` in the `freeda-0.1` subdirectory for instructions on compiling a new element into *f*REEDA™.

3.3 Template Files for new Nonlinear Elements

The file listings in this section are provided to simplify the creation of new elements. All that is required is to replace `Element1` by the name of the actual class, copy the files with appropriate names in the `elements/` directory and write the code to implement the `init` and the `eval` functions, and the function `getLocalRefIdx` if you want to create an element with multiple reference nodes.

3.3.1 Header File

The header file template is shown in Listing 3.18.

Listing 3.18: Element1.h

```
1
2 // This may look like C code, but it is really -*- C++ -*-
3 //
4 // This is a generic element template
5 //
6 //              +-----+
7 //          o---+     +---o
```

```
8  //              +-----+
9  //
10
11 #ifndef Element1_h
12 #define Element1_h 1
13
14 class Element1 : public AdolcElement
15 {
16 public:
17
18   Element1(const string& iname);
19
20   ~Element1() {}
21
22   static const char* getNetlistName()
23   {
24     return einfo.name;
25   }
26
27   // Do some local initialization
28   virtual void init() throw(string&); // to perform initialization that
29                                       // cannot be done by the constructor
30                                       // and to create the Adolc Tape
31
32   // Implement the getLocalRefIdx function if you are creating
33   // an element with multiple reference nodes.
34   // Get a vector with the indexes of the local reference nodes.
35   // Get terminal pointers in term_list
36   // ordered by local reference node:
37   //
38   // t0 t1 t2 t3
39   //
40   //   ^    ^
41   //  LRN1 LRN2
42   //
43   // local_ref_vec contains: {1, 3}
44   //
45   virtual void getLocalRefIdx(UnsignedVector& local_ref_vec, // to implement a spatially
46                               TerminalVector& term_list);  // distributed element, or an
47                                                            // electro-thermal one with
48                                                            // multiple reference nodes.
49
50 private:
51
52   // Implement the eval function. It is here where you write
53   // your device equations.
54   virtual void eval(adoublev& x, adoublev& vp, adoublev& ip);
55
56   // Parameter variables (replace by your parameter variables here)
57   double par1, par2, ..., parn;
58
59   // Element information
60   static ItemInfo einfo;
61
62   // Number of parameters of this element
63   static const unsigned n_par;
64
65   // Parameter information
66   static ParmInfo pinfo[];
67
68 };
69
```

```
70 #endif
```

3.3.2 Class Source File

The class source file template is shown in Listing 3.19.

Listing 3.19: Element1.cc

```
1
2 #include "../network/ElementManager.h"
3 #include "../network/AdolcElement.h"
4 #include "../analysis/FreqMNAM.h"
5 #include "../analysis/TimeMNAM.h"
6 #include "Element1.h"
7
8 // Static members (set to the number of parameters of your element)
9 const unsigned Element1::n_par = ;
10
11 // Element information
12 ItemInfo Element1::einfo = {
13   "tres",
14   "Element1",
15   "Your Name",
16   "http://your.web.page/element1.html"
17 };
18
19 // Parameter information
20 ParmInfo Element1::pinfo[] = {
21   {"par1", "Parameter 1 (Ohms)", TR_DOUBLE, true},
22   {"par2", "Parameter 2 (The Unit)", TR_DOUBLE, false}
23       .                                                    // Note that the sequence of
24       .                                                    // parameters here should be
25       .                                                    // the same as in the constructor
26   {"parn", "Parameter n (The Unit)", TR_BOOLEAN, false}
27 };
28
29 Element1::Element1(const string& iname) : AdolcElement(&einfo, pinfo, n_par, iname)
30 {
31   // Set parameters
32   paramvalue[0] = &par1;
33   paramvalue[1] = &(par2 = 1e-12); // Example of how to set a default value.
34         .
35         .                              // Note that the sequence of parameters here
36         .                              // should be the same as in the struct pinfo[]
37   paramvalue[n] = &(parn = false);
38 }
39
40 void Element1::init() throw(string&)
41 {
42     // Set the number of terminals (Set to the number of terminals of
43     // your element)
44     setNumTerms(4);
45
46     // Set flags. If your element has multiple reference nodes,
47     // set the second flag to MULTI_REF
48     setFlags(NONLINEAR | ONE_REF | TR_TIME_DOMAIN);
49
50     // Set number of states
51     setNumberOfStates(2);
52
53     // create the tape
54     IntVector var(2,0);
```

```
55      var[1] = 1;
56      createTape(var);
57  }
58
59  void Element1::eval(adoublev& x,
60                      adoublev& vp, adoublev& ip)
61  {
62    // Implement the eval function
63    // write the device equations
64  }
65
66  void Element1::getLocalRefIdx(UnsignedVector& local_ref_vec,
67                                TerminalVector& term_list)
68  {
69      // Make sure the vectors are empty
70      // always write these functions as they are
71      term_list.erase(term_list.begin(), term_list.end());
72      local_ref_vec.erase(local_ref_vec.begin(), local_ref_vec.end());
73
74      // Insert vector elements
75      // push_back all the terminals in the term_list
76      // push_back all the local reference terminal in the local_ref_vec
77      term_list.push_back(getTerminal(0));
78      term_list.push_back(getTerminal(1)); // Local reference terminal
79      local_ref_vec.push_back(1); // Local reference index
80      term_list.push_back(getTerminal(2));
81      term_list.push_back(getTerminal(3)); // Local reference terminal
82      local_ref_vec.push_back(3); // Local reference index
83  }
```

3.4 A Note on the Eval Routines in fREEDA™

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions with m components in n real or complex variables. fREEDA™ uses the package ADOL-C which automatically computes the derivatives of non-linear functions by successive implementation of the chain rule. These calculation occur at a fraction of the time and is relatively free of truncation errors. The interface details to the ADOL-C library from within fREEDA™ can be found in path_to_freeda/simulator/network/NAdolcElement.cc.

Every nonlinear element in fREEDA™ has an eval function that describes the constitutive equations for the particular element at any instant. This function is called from within NAdolcElement.cc during evaluation of the tape. For more details on the tape, refer to ADOL-C documentation.

3.4.1 Parameterized Device Models

A non-linear device model can be described with the following set of equations

$$\mathbf{v(t)} = \mathbf{v}(\mathbf{x(t)}, \mathbf{dx/dt}, \ldots, \mathbf{d^m x/dt^m}, \mathbf{x_D(t)}) \tag{3.1}$$

$$\mathbf{i(t)} = \mathbf{i}(\mathbf{x(t)}, \mathbf{dx/dt}, \ldots, \mathbf{d^m x/dt^m}, \mathbf{x_D(t)}) \tag{3.2}$$

where $\mathbf{v}$(t) and $\mathbf{i}$(t) are vectors of voltages and currents at the ports of the non-linear device, $\mathbf{x}$(t) is a vector of parameters or state variables and $\mathbf{x_D}$(t) is a vector of time delayed state variables. There are several models like the microwave diode, the Gummel-Poon BJT and Berkeley's BSIM4 which for accurate modeling, require charge to be defined as a state variable. In some cases it may not be possible to find the voltage as a function of charge and the only alternative is to treat charge as a state variable.

Consider a simplified model for the microwave diode shown in Figure 3.2. The corresponding equations for this model are

$$i(v) = I_s(\exp(\alpha v) - 1) \tag{3.3}$$

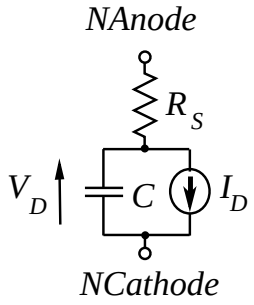

Figure 3.2: Simplified schematic for a diode model

and

$$c_j(v) = \quad C_{t0}(1 - v/\phi)^{-\gamma} + C_{d0}\exp(\alpha' v) \quad if\ v \le .8\phi \tag{3.4}$$

$$C_{t0}(.2)^{-\gamma} + C_{d0}\exp(\alpha' v) \qquad if\ v \ge .8\phi \tag{3.5}$$

where v is the junction voltage. The capacitor voltage q_j is given by

$$q_j(v) = \int_0^v c_j(u)du \tag{3.6}$$

Accurate transient analysis requires q_j to be chosen as a state variable. We need v to calculate $i_1(v)$. Since it is not possible to solve analytically for $v(q)$, the alternative is to model the diode with two state variables, namely v and q . The diode equations can now be formulated in two stages. Hence for this case, we will have two *eval* functions; *eval*1 and *eval*2 . In *eval*1, we will calculate $i(v)$ and $q(v)$, both are which are functions of voltage v. In this case, the values of $q(v)$ are passed to *eval*2, which automatically calculates dq/dt. Hence we now have the current through the capacitor as

$$i_c = \frac{dq}{dt} \tag{3.7}$$

All the code that performs the actual derivatives is outside the nonlinear model and is handled automatically. Currently in fREEDA™, only three levels of eval are supported. This means that the highest order of derivatives possible is two. Most nonlinear elements do not require more than two levels of derivatives. The entire formulation can be generalized as follows:

$$\begin{aligned} \text{stage 1} \quad : \quad & f_1(x, x_D) \\ & g_1(x, x_D) \end{aligned} \tag{3.8}$$

$$\begin{aligned} \text{stage 2} \quad : \quad & f_2(f_1, dg_1/dt) \\ & g_2(f_1, dg_1/dt) \end{aligned} \tag{3.9}$$

$$\vdots$$

$$\begin{aligned} \text{stage n} \quad : \quad & v(f_{n-1}, dg_{n-1}/dt) \\ & i(f_{n-1}, dg_{n-1}/dt) \end{aligned} \tag{3.10}$$

3.5 Contributors

The following contributed to this chapter:

Frank P. Hart

Houssam S. Kanj

Nikhil Kriplani

Carlos Christoffersen

Michael Steer.

Appendix A

Object Oriented Programming Basics

Object oriented programming (OOP) [61] provides a means for abstraction in both programming and design. OOP does not deal with programming in the sense of developing algorithms or data structures but it must be studied as a means for the organization of programs and, more generally, techniques for designing programs.

As the primary means for structuring a program or design, OOP provides *objects*. Objects may model real life entities, may function to capture abstractions of arbitrary complex phenomena, or may represent system artifacts such as stacks or graphics. Operationally, objects control the computation. From the perspective of program development, however, the most important characteristic of objects is not their behavior as such, but the fact that the behavior of an object may be described by an abstract characterization of its interface. Having such a characterization suffices for the design. The actual behavior of the object may be implemented later and refined according to the need. A *class* specifies the behavior of the objects which are its instances. Also, classes act as templates from which actual objects may be created. An *instance* of a class is an object belonging to that class. A procedure (or function) inside an object is called a *method*. A *message* to an object is a request to execute a method.

Inheritance is the mechanism which allows the reuse of class specifications. The use of inheritance results in a class hierarchy that, from an operational point of view, decides what is the method that will be selected in response to a message.

Finally, an important feature of OO languages is their support for *polymorphism*. This makes it possible to hide different implementations behind a common interface.

The notion of flow diagram in procedural programming is replaced in OOP by a set of objects which interact by sending messages to each other.

We will briefly review what are traditionally considered to be features and benefits of OOP. Both *information hiding* (also known as *encapsulation*) and *data abstraction* relieve the task of the programmer using existing OO code, since with these mechanisms the programmer's attention is no longer distracted by irrelevant implementation details. The flexible dispatching behavior of objects that lends them their polymorphic behavior is due to the dynamic binding of methods to messages. For the C++ language, polymorphic object behavior is effected by using *virtual functions*, for which in contrast to ordinary functions, the binding to an actual function takes place at run time and not at compile time.

Encapsulation promotes *modularity*, meaning that objects may be regarded as the building blocks of a complex system. Another advantage often attributed to the OOP is code reuse. Inheritance is an invaluable mechanism in this respect, since it enables the programmer to modify the behavior of a class of objects without requiring access to the source code.

Although an object oriented approach to program development indeed offers great flexibility, some of the problems it addresses are intrinsically difficult and cannot really be solved by mechanisms alone. For example, it is more likely to achieve a stable modularization when shifting focus from programming to design.

C++ virtual functions [71] can have big performance penalties such as extra memory accesses or the

possibility of an unpredictable branch so pipelines can grind to a halt (note however, that some architectures have branch caches which can avoid this problem). There are several research projects which have demonstrated success at replacing virtual function calls with direct dispatches. Note however, that virtual functions are not always bad when it comes to performance. The important questions are: How much code is inside the virtual function? How often is it used? If there is a lot of code (*i.e.* more than 25 flops), then the overhead of the virtual function will be insignificant. But if there is a small amount of code and the function is called very often (*e.g.* inside a loop), then the overhead can be critical.

A.1 UML diagrams

The Unified Modeling Language (UML) [84, 66] is a language for specifying, visualizing, and constructing the artifacts of software systems as well as for business modeling. The goal of UML is to become a common language for creating models of object oriented computer software. It is used here to graphically illustrate the relationship of classes using what is called a class diagram. A class diagram is a graph of Classifier elements with connections indicating by their various static relationships (Figure 1.1). (Note that a "class" diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structural diagram" but "class diagram" is shorter and its use is well established.) A class is drawn as a solid-outline rectangle with 3 compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations. Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it.

Each instance of type **Element**, for example, seems to contain an instance of type **ElementData**. This class relationship is indicated by the joining line. The relationship is composition — indicated by the solid diamond symbol. The arrowhead denotes that the relationship is navigable in only one direction, *i.e.*, **ElementData** does not know anything about **Element**. The inheritance relationship in UML is depicted by the triangular arrowhead and points to the base class. A line from the base of the arrowhead connects it to the derived classes, e.g. **Element** is derived from **NetListItem**.

Other forms of containment do not have whole/part implications and are called association relationships indicated by a line drawn between the participating classes. (This relationship will almost certainly be implemented using pointers unless it is very weak.) If the relationship between two classes is very weak (*i.e.* very little data is shared) then a dashed line is used. For example, in Figure 1.1, **ElementManager** somehow depends upon **Diode**. (In C++ the weak relationship is almost always implemented using an `#include`.)

An illustration showing examples for the notation is given in Figure A.1.

A.1.1 Collaboration Diagrams

The collaboration diagram [84] is part of the Unified Modeling Language (UML), a language for specifying, visualizing, and constructing the artifacts of software systems as well as for business modeling. The UML represents a collection of 'best engineering practices' that have proven successful in the modeling of large and complex systems. Behavior is implemented by sets of objects that exchange messages within an overall interaction to accomplish a purpose. To understand the mechanisms used in a design, it is important to see only the objects and the messages involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are part for other purposes. Such a static construct is called a collaboration. A collaboration is a set of participants and relationships that are meaningful for a given set of purposes. The identification of participants and their relationships does not have global meaning. In the collaboration diagram, each box represents an object (in this case, either a terminal or an element). The lines between boxes represent associations and the arrows are messages. The order in which messages are sent is shown by the numbers.

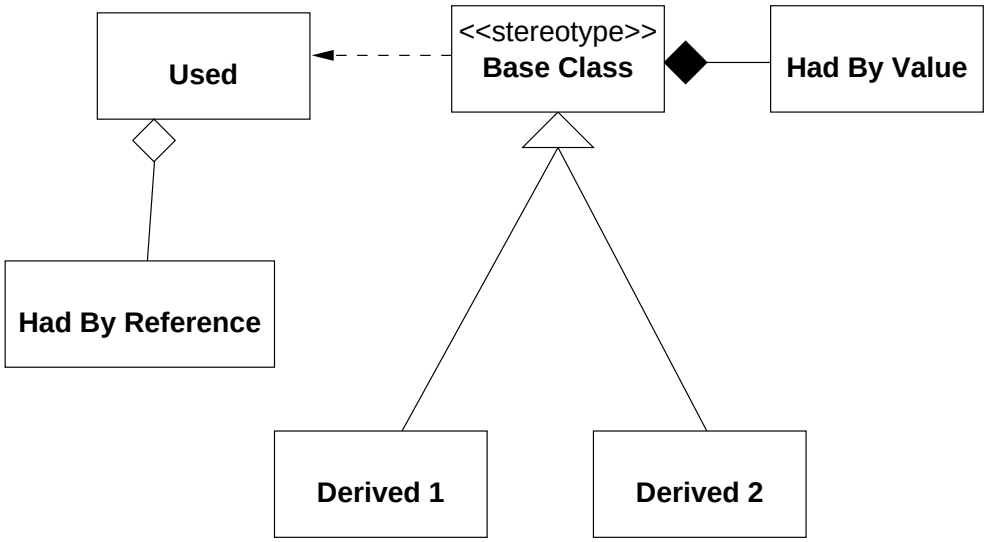

Figure A.1: Notation for a class diagram.

A.2 Contributors

The following contributed to this chapter

Carlos Christoffersen

Michael Steer.

Appendix B

Release Notes

Version 1.0.0
Initial release.

Version 1.1.0
Noted that fREEDA™ is now a registered trademark in the front matter.
Added Section C on supporting software libraries.
Minor corrections were made to Section 2.
Major changes and additions were made to Section 3, including new listing formatting, details on createTape(), relationship between terminals and state variables, and more.
Added Appendix D on the Netlist Format.

Appendix C

Support Libraries

A large number of support libraries are available (many of them freely) and some of these are used in Transim. The various libraries, which should be of general interest to the microwave modeling community, are described below.

C.1 Solution of Sparse Linear Systems (Sparse, SuperLU)

Sparse 1.3[1] [74] is a flexible package of subroutines written in C used to numerically solve large sparse systems of linear equations. The package is able to handle arbitrary real and complex square matrix equations. Besides being able to solve linear systems, it is also able to quickly solve transposed systems, find determinants, and estimate errors due to ill-conditioning in the system of equations and instability in the computations. Sparse also provides a test program that is able to read matrix equation from a file, solve it, and print useful information (such as condition number of the matrix) about the equation and its solution. Sparse was originally written for use in circuit simulators and is well adapted to handling nodal- and modified-nodal admittance matrices.

SuperLU[2] is used in the wavelet and time marching transient analyses. It contains a set of subroutines to numerically solve a sparse linear system $\mathbf{Ax} = \mathbf{b}$. It uses Gaussian elimination with partial pivoting (GEPP). The columns of $\mathbf{A}$ may be preordered before factorization; the preordering for sparsity is completely separate from the factorization. SuperLU is implemented in ANSI C. It provides support for both real and complex matrices, in both single and double precision.

C.2 Vectors and Matrices (MV++, MTL)

Most of the vector and matrix handling in *f*REEDA™ uses *MV++*[3] [73]. This is a small set of vector and simple matrix classes for numerical computing written in C++. It is not intended as a general vector container class but rather designed specifically for optimized numerical computations on RISC and pipelined architectures which are used in most new computer architectures. The various MV++ classes form the building blocks of larger user-level libraries. The MV++ package includes interfaces to the computational kernels of the Basic Linear Algebra Subprograms package (BLAS) which includes scalar updates, vector sums, and dot products. The idea is to utilize vendor-supplied, or optimized BLAS routines that are fine-tuned for particular platforms.

The *Matrix Template Library* (MTL)[4] is a high-performance generic component library that provides comprehensive linear algebra functionality for a wide variety of matrix formats. It is used in the wavelet

[1] http://www.netlib.org/sparse/
[2] http://www.nersc.gov/xiaoye/SuperLU/
[3] http://math.nist.gov/mv++/
[4] http://www.lsc.nd.edu/research/mtl/

and time marching transient analyses.

As with STL, MTL uses a five-fold approach, consisting of generic functions, containers, iterators, adaptors, and function objects, all developed specifically for high performance numerical linear algebra. Within this framework, MTL provides generic algorithms corresponding to the mathematical operations that define linear algebra. Similarly, the containers, adaptors, and iterators are used to represent and to manipulate matrices and vectors.

C.3 Solution of Nonlinear Systems (NNES)

Nonlinear systems of equations in *f*REEDA™ are solved using the *NNES*[5] [59] library. This package is written in Fortran and provides Newton and quasi-Newton methods with many options including the use of analytic Jacobian or forward, backwards or central differences to approximate it, different quasi-Newton Jacobian updates, or two globally convergent methods, etc. This library is used through an interface class (**NLSInterface**), so it is possible to install a different routine to solve nonlinear systems if desired by just replacing the interface (four different nonlinear solvers have already been used). The Fortran routine *NLEQ1* (Numerical solution of nonlinear (NL) equations (EQ)[6]) can also be used as a compile option.

C.4 Fourier Transform (FFTW)

Fourier transformation is implemented in *f*REEDA™ using the *FFTW*[7] library [52]. FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. Benchmarks, performed on a variety of platforms show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is portable: the program performs well on most computer architectures without modification.

C.5 Automatic Differentiation (Adol-C)

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions with m components in n real or complex variables [72]. Often these functions are defined by sequential evaluation procedures involving many intermediate variables. By eliminating the intermediate variables symbolically, it is theoretically always possible to express the m dependent variables directly in terms of the n independent variables. Typically, however, the attempt results in unwieldy algebraic formulae, if it can be completed at all. Symbolic differentiation of the resulting formulae will usually exacerbate this problem of *expression swell* and often entails the repeated evaluation of common expressions.

An obvious way to avoid such redundant calculations is to apply an optimizing compiler to the source code that can be generated from the symbolic representation of the derivatives in question. Exactly this approach was investigated by Speelpenning during his Ph.D. research [75] at the University of Illinois from 1977 to 1980. Eventually he realized that at least in the cases $n = 1$ and $m = 1$, the most efficient code for the evaluation of derivatives can be obtained directly from the evaluation of the underlying vector function. In other words, he advocated the differentiation of evaluation algorithms rather than formulae. In his dissertation he made the particularly striking observation that the gradient of a scalar-valued function (*i.e.* $m = 1$) can always be obtained with no more than five times the operations count of evaluating the function itself. This bound is completely independent of n, the number of independent variables, and allows the row-wise computation of Jacobians for at most $5m$ times the effort of evaluating the underlying vector function.

[5] http://www.netlib.org/opt/

[6] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). Contact: Lutz Weimann, ZIB, Division Scientific Computing, Department Scientific Software, e-mail: weimann@zib.de

[7] http://www.fftw.org

Given a code for a function $F : \Re^n \rightarrow \Re^m$, automatic differentiation (AD) uses the chain rule successively to compute the derivative matrix. AD has two basic modes, forward mode and reverse mode [76]. The difference between these two is the way the chain rule is used to propagate the derivatives.

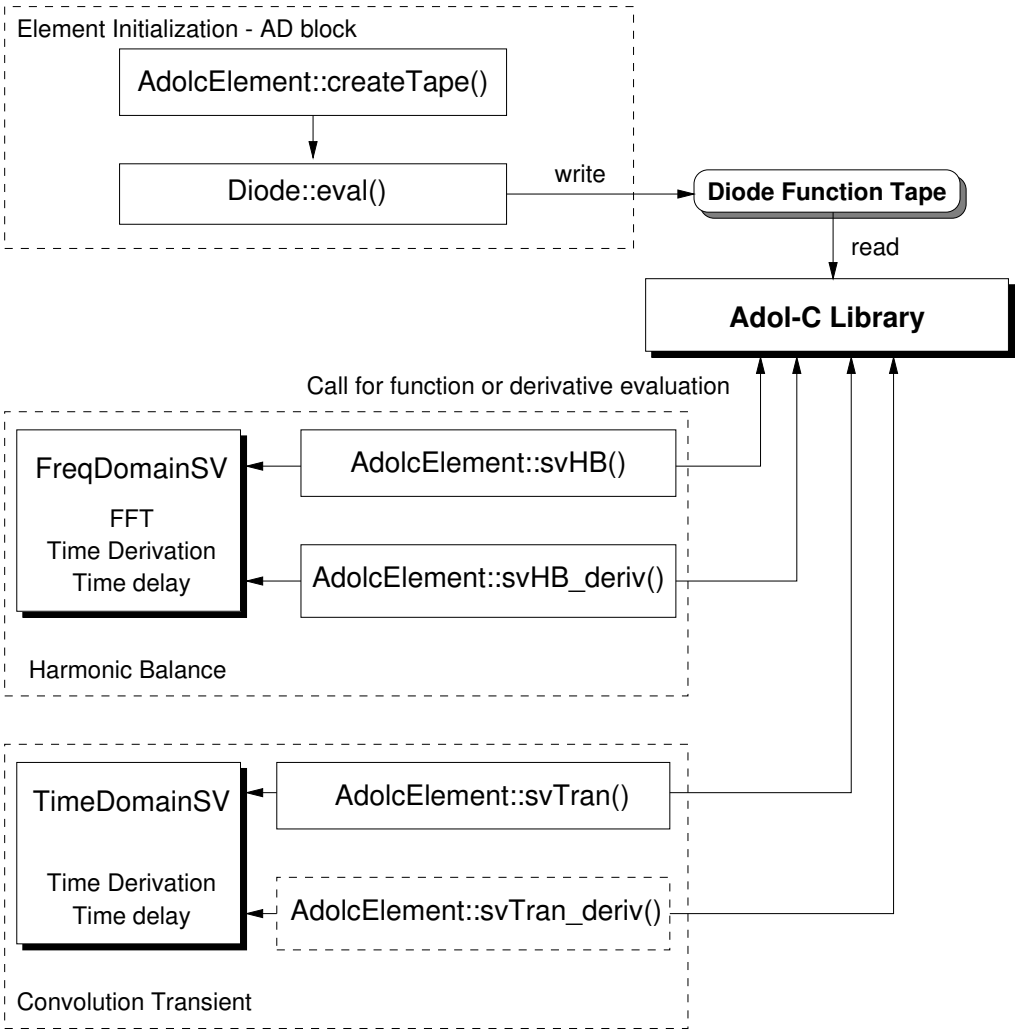

Figure C.1: Implementation of automatic differentiation.

A versatile implementation of the AD technique is *Adol-C*[8] [72], a software package written in C and C++. The numerical values of derivative vectors (for example, required to fill a Jacobian in Harmonic Balance analysis [2], see Figure C.1) are obtained free of truncation errors at a small multiple of the run time required to evaluate the original function with little additional memory required. It is important to note that AD is not numerical differentiation and the same accuracy achieved by evaluating analytically developed derivatives is obtained.

The `eval()` method of the nonlinear element class is executed at initialization time and so the operations to calculate the currents and voltages of each element are recorded by Adol-C in a *tape* which is actually an internal buffer. After that, each time that the values or the derivatives of the nonlinear elements are required, an Adol-C function is called and the values are calculated using the tapes. This implementation

[8] http://www.math.tu-dresden.de/ adol-c/

is efficient because the taping process is done only once (this almost doubles the speed of the calculation compared to the case where the functions are taped each time they are needed). When the Jacobian is needed, the corresponding Adol-C function is called using the same tape. We have tested the program with large circuits with many tones, and the function or Jacobian evaluation times are always very small compared with the time required to solve the matrix equation (typically some form of Newton's method) that uses the Jacobian. The conclusion is that there is little detriment to the performance of the program introduced by using automatic differentiation. However the advantage in terms of rapid model development is significant. The majority of the development time in implementing models in simulators, particularly harmonic balance simulators, is in the manual development of the derivative equations. Unfortunately the determination of derivatives using numerical differences is not sufficiently accurate for any but the simplest circuits. With Adol-C full 'analytic' accuracy is obtained and the implementation of nonlinear device models is dramatically simplified. From experience the average time to develop and implement a transistor model is an order of magnitude less than deriving and coding the derivatives manually. Note that time differentiation, time delay and transformations are left outside the automatic differentiation block. The calculation speed achieved is approximately ten times faster than the speed achieved by including time differentiation, time delay and transformations inside the block.

C.6 Contributors

The following contributed to this chapter

Carlos Christoffersen

Michael Steer.

Appendix D

Netlist Format

The netlist input of TRANSIM is almost compatible to Spice. There are a number of additional features and these are documented below. The focus of the additions is to facilitate the addition of new models, allow variables, and support hierarchical descriptions of coupling in a network.

D.1 STRUCTURE OF TRANSIM'S NETLIST

The transim netlist mainly consists in a title, an analysis specification, a list of connected elements[1], and a list of output commands.

D.1.1 LEXICAL

TRANSIM's grammatical rules are very similar to those of spice:

whitespace blank
a newline followed immediately by a + sign. a tab
a vertical tab
a newpage

identifier A character sequence beginning with an Alphabetic character

$$A - Za - z$$

variables A variable must begin with an alphabetic character or a $ followed by alphanumeric characters or '_' or '.'

Example:

```
HEIGHT
\$height
height.1_1
```

Note that `HEIGHT` and `height` are identical as case is not preserved.

strings Either as an identifier (a continuous sequence of alphanumeric characters or enclosed within double quotes.
The following special escaped characters are allowed in strings defined within double quotes.

"To include a double quote in a string.

[1] element: a model of a physical component of a network.

`n`To indicate a newline

Examples:

```
gate
"VOLTAGE WAVEFORM"
```

Note: Strings may continue across lines using the Spice continuation syntax:

```
"VOLTAGE
+ WAVEFORM"
```

or simply by continuing across a line as in

```
"VOLTAGE
 WAVEFORM"
```

numbers "E" or "e" to indicate exponent.

dotted command A "." folowed by alphabetic characters

lf A line feed or cariage return.

CAPITALIZATION

The case of identifiers and keywords is ignored in TRANSIM netlists. The significance of case is retained only within quoted strings, and in that case it is always retained. Internally characters are mapped to lower case.

D.1.2 CONTINUATION OF LINE

A line beginning with a plus sign is considered to be the continuation of the previous one.

D.1.3 TITLE LINE

```
*** Unit Cell Folded Slot Antenna ***
```

As in Spice, the first line of the netlist file is the title and does not contain commands.

D.1.4 COMMENTS

```
* Local reference nodes
```

As in Spice, comment lines begin with an asterisk.

D.1.5 .options

Used to set up quantities similar to spice syntax. The general syntax is

`.options <`*identifier*`> = <`*value*`>`

All identifiers set in a .options card are treated as a variable. *value* may be an number or a previously defined variable.

Some variables are preset:

variable	default
defl	OPTIONS_DEFAULT_DEFL
defw	OPTIONS_DEFAULT_DEFW
defad	OPTIONS_DEFAULT_DEFAD
defas	OPTIONS_DEFAULT_DEFAS
tnom	OPTIONS_DEFAULT_TNOM
numdgt	OPTIONS_DEFAULT_NUMDGT
cptime	OPTIONS_DEFAULT_CPTIME
limpts	OPTIONS_DEFAULT_LIMPTS
itl1	OPTIONS_DEFAULT_ITL1
itl2	OPTIONS_DEFAULT_ITL2
itl4	OPTIONS_DEFAULT_ITL4
itl5	OPTIONS_DEFAULT_ITL5
reltol	OPTIONS_DEFAULT_RELTOL
trtol	OPTIONS_DEFAULT_TRTOL
abstol	OPTIONS_DEFAULT_ABSTOL
chgtol	OPTIONS_DEFAULT_CHGTOL
vntol	OPTIONS_DEFAULT_VNTOL
pivrel	OPTIONS_DEFAULT_PIVREL
gmin	OPTIONS_DEFAULT_GMIN

The defaults are defined in spice.h

D.1.6 .model

```
.model c_line tlinp4 ( z0mag=75.00 k=7 fscale=1.e10
+ alpha = 59.9 )
```

Each `.model` is a statement that associates a name (<*model name*>) with a list of parameter values (<*param-list*>). The parameter names given must be the names of parameters of the element specified after the model keyword. Thus, `alpha` and `z0mag` must be parameters of `tlinp4` in the above example.

Further, the values assigned to parameters must be of an appropriate type. The parser goes to some lengths to coerce types where the result is sensible (i.e., if you give an integer value "1" to a parameter of float type, the parameter will be assigned the floating-point value "1.0"). However, you can't assign string values to float parameters, or vice-versa.

The `.model` statements define the default characteristics of the different physical elements ("models") in a network.

The syntax is as for spice

`.model` <*model name*> <*type name*> ([<*parameter name*> = <*value*>]*)

Here, <*model name* > is an identifier by which the model is referred to. <*type name*> is the physical element name that the model refers to. the *parameter name* must be a valid parameter for the element (indicated by <*type name*>) referred to.

Any number of models may be specified for a single element.

D.1.7 Analysis Specification

```
.ac start = 3.6GHz stop = 4.8GHz n_freqs = 7
```

This line consists in a dot followed by the analysis name and a list of parameters. See the analysis catalog for a list of analysis and the description of the paramenters.

Note that 4.8GHz is equivalent to 4.8e9 or 4.8g. This is the same as in Spice.

D.1.8 Element Specification

```
nport:cpw_2 10 20 100 200 filename = "unitcell.yp"
```

Elements are specified with the element type name (`nport` in this example) followed by a colon and the element instance name. Then a list of nodes (or terminals) to which the element is connected followed by a paramenter list. See the element catalog for a list of available elements and the description of the paramenters.

The terminals can be named using integer numbers or strings. When using strings, they must be enclosed in quotes.

Regular passive elements (R, L and C) also support the standard Spice syntax with the following additions common to all elements in Transim:

1. A `.model` specification is allowed for all elements.
2. Anything that can appear in a `.model` specification can be included in the specification of the element.
3. If a parameter is not specified either through an element specification or a `.model` specification then the default parameters for that model will apply to this element.

D.1.9 END OF NETLIST

Every netlist must finish with a `.end` control card.

D.1.10 SUBCIRCUITS

The subcircuit definition and instantiation is pretty much as in Spice. The definition may appear after or before the instatiation in the netlist. Node names inside the subcircuit are local to the subcircuit. The following is an example for a three-terminal subcircuit.

```
...

.subckt era6 1 5 "gnd1"

... (subcircuit definition)

.ends

...

xamp1 10 50 0 era6

...
```

The name of the subcircuit instance must begin with `x`.

D.2 OUTPUT CONTROL

Transim has an interpretive output language which uses a reverse polish notation syntax. The operators operate on a stack and as an operation is performed zero or more arguements are consumed by an opertor. This is an extremely powerful way of controlling output.

OUTPUT COMMANDS

`.out write ( (<`*qualifier*`> <`*value*`>* )* <`*operator*`> )* in <`*filename*`>`

or

`.out plot ( (<`*qualifier*`> <`*value*`>* )* <`*operator*`> )* [[<`*gnuplotPostambleScript*`>] <`*gnuplotPreambleScript*`>] in <`*filename*`>`

or

`.out system <string>`

D.2.1 WRITING

`.out write ( (<qualifier> <value>* )* <operator> )* in <filename>`

The `write` command writes what is left on the stack into the file *filename*.

EXAMPLE

```
.out write term 4 vt in "4v.out"
```

This writes the time domain voltage at terminal 4 using the file 4v.out as an output file.

D.2.2 PLOTTING

`.out plot ( (<qualifier> <value>* )* <operator> )* [[<gnuplotPostambleScript>] <gnuplotPreambleScript>] in <filename>`

The `plot` command writes what is left on the stack into the file *filename* and initiates a plot. The file can be plotted interactively using the Transim Output Viewer. Also, a file named `<filename>.cmd` is created. This file is a gnuplot [?] script file that plots the desired data. The Scripts are optional strings and are used to send additional commands to the gnuplot program.

<gnuplotPreambleScript> is a string of semicolon delineated gnuplot commands prior to the plot command which is automatically issued.

<gnuplotPostambleScript> is a string of semicolon delineated gnuplot commands after the plot command.

If the option `gnuplot` is present in the `.options` card, the gnuplot program will be called automatically by Transim. Note that this is generally not needed when using the Output Viewer.

Example

EXAMPLE

```
.out plot term 4 vt in "4v.out"
```

There are no script commands here. This plots the time domain voltage at terminal 4 using the file 4v.out as an output file. This functions as both a write and a plot.

D.2.3 RUNNING A SYSTEM COMMAND

`.out system <string>`

Use this to run the string as a command to the operating system.

EXAMPLE

```
.out system "echo Hello"
```

Prints “Hello” on the screen.

D.2.4 NOMENCLATURE

The following nomenclature is used in describing the output operators.

type	description
scalar numeric types	
i	integer
f	floating-point
r	real (integer or floating-point)
c	complex
s	scalar (integer, floating-point or complex)
scalar and mixed numeric types	
fv	floating-point vector
cv	complex vector
v	floating-point or complex vector
fsv	floating-point scalar or vector
csv	complex scalar or vector
sv	scalar or vector (any)
prom	an appropriately-promoted numeric type
-x	(suffix to vector types) x data required
other types	
any	any type
string	character string
var	variable name
file	data file
func	function pointer

D.2.5 QUALIFIERS

type	description
qualifiers (network types)	
term	terminal (or node)
element	circuit element

D.2.6 Operators

OPERATORS

General operators

GENERAL OPERATORS

operator	function	argument(s)	result
`dup`	duplicate object	*any*	*same*
`get`	get element of vector	*arg:v* *index:i*	*s*
`put`	modify element of vector	*arg:v* *index:i* *val:s*	*v*
`stripx`	remove x data	*vx*	*v*
`pack`	concatenates last *vx*'s on stack	variable number of *vx*	*m*
`system`	execute shell command	*string*	*none*

D.2.7 Network Operators

`vf`	complex freq. domain voltage vector at a terminal	*term*	*cv*
`if`	complex freq. domain current vector at a terminal	*term*	*cv*
`xf`	complex freq. domain state variable vector at a terminal	*term*	*cv*
`vt`	time domain voltage vector at a terminal	*term*	*fv*
`it`	time domain current vector at a terminal	*term*	*fv*
`ut`	time domain voltage vector at an element port	*elem*	*fv*

RPN ARITHMETIC OPERATORS

Arithmetic Operators for reverse polish notation e.g. 3 4 add = 7

`add`	addition	*sv* *sv*	*prom*
`sub`	subtraction	*sv* *sv*	*prom*
`mult`	multiplication	*sv* *sv*	*prom*
`div`	division	*sv* *sv*	*prom*
`real`	real part	*csv*	*fsv*
`imag`	imaginary part	*csv*	*fsv*
`mag`	magnitude	*csv*	*fsv*
`abs`	absolute value or magnitude	*sv*	*fsv*
`contphase`	continuous phase	*csv*	*fsv*
`prinphase`	principal value phase	*csv*	*fsv*
`conj`	complex conjugate	*csv*	*csv*
`neg`	additive inverse (negative)	*sv*	*sv*
`recip`	reciprocal	*sv*	*sv*

D.2.8 Conventional arithmetic operators

CONVENTIONAL ARITHMETIC OPERATORS Conventional Arithmetic Operators e.g. 3 + 4 = 7 Not fully implemented. Do not use and reserved for future expansion.

`+`	addition	*sv* *sv*	*prom*
`-`	subtraction	*sv* *sv*	*prom*
`*`	multiplication	*sv* *sv*	*prom*
`/`	division	*sv* *sv*	*prom*
`^`	poer	*sv* *sv*	*prom*
`**`	poer	*sv* *sv*	*prom*

Mathematical operators

MATHEMATICAL OPERATORS

`db`	dB ($20\log_{10}$)	*sv*	*fsv*
`db10`	dB applied to power ($10\log_{10}$)	*sv*	*fsv*
`rad2deg`	convert radians to degrees	*fsv*	*fsv*
`deg2rad`	convert degrees to radians	*fsv*	*fsv*
`minlmt`	limit the minimum value	*arg:fsv* *min:f*	*fsv*
`maxlmt`	limit the maximum value	*arg:fsv* *max:f*	*fsv*
`diff`	differences	*fsv*	*fsv*
`deriv`	derivative	*fsv*	*fsv*
`sum`	sums	*fsv*	*fsv*
`integ`	integral	*fsv*	*fsv*

Signal processing operators

SIGNAL PROCESSING OPERATORS

`smpltime`	current analysis timebase as x *and* y of result	*none*	*fv*
`sweepfrq`	current analysis sweep frequencies as x *and* y of result	*none*	*fv*
`smplcvt`	interpolate *signal1* over timebase of *signal2*	*signal1:v* *signal2:vx*	*vx*
`sweepcvt`	interpolate *frq1* over sweep frequencies of *frq2*	*frq1:v* *frq2:vx*	*vx*
`maketime`	create timebase starting at $t = 0$ in x *and* y of result	*tmax:r* *pts:i*	*vx*
`makesweep`	create sweep frequencies starting at $f = 0$ in x *and* y of result	*fmax:r* *pts:i*	*vx*
`fft`	FFT (argument should have 2^k points)	*timedata:fv*	*cv*
`invfft`	inverse FFT (argument should have $2^k - 1$ points)	*frqdata:cv*	*fv*
`cconv`	real circular (FFT) convolution with zero padding	*signal1:fv* *signal2:fv*	*fv*
`upcconv`	unpadded real circular (FFT) convolution	*signal1:fv* *signal2:fv*	*fv*
`sconv`	slow (time-domain) real convolution	*signal1:fv* *signal2:fv*	*fv*
`fconv`	fast (approximate) real convolution	*signal1:fv* *signal2:fv*	*fv*
`lpbwfrq`	lowpass Butterworth filter frequency response	*frqvec:vx* *corner:f* *order:i*	*cvx*

D.3 EXAMPLE: SIMULATION OF A FOLDED SLOT ANTENNA

The netlist format is illustrated using an example. This example uses local reference nodes. For a discussion of the local reference node concept see chapter **??**. Transim provides the local references as a convenience tool, but it is still possible to treat circuits in a conventional way using the node "0" or "`gnd`" as a global reference.

EM modeling yields a port-based y parameters of the antennas at each frequency of interest. The transfer of data between the EM and circuit simulators (typically a file) includes a header with port grouping information (a port grouping includes terminals associated with a specific local reference node). This is required by the circuit simulator in order to expand the port-based matrix into nodal form and also to check the connectivity of the spatially-distributed circuit.

Below is shown the data file for this example. Each port belong to a different group, so the element has four terminals. After reading the header the circuit simulator knows the number of elements of the matrix and the port number and local reference corresponding to each row and column of the matrix.

```
# port:group
1:1
1:2
# GHZ Y RI R 50
4
0.00355603      -0.0233196
-0.00121905     -0.00496212
-0.00121905     -0.00496212
0.00355603      -0.0233196
...
```

The rest of the file consist in a list of frequencies and the associated matrix elements in complex form.

The parser provides several facilities such as the `.model` statement support for any element type and a complete reverse polish notation calculator for the output data. The corresponding netlist is shown below.

```
*** Unit Cell Folded Slot Antenna ***

.ac start = 3.6GHz stop = 4.8GHz n_freqs = 7

* Local reference nodes
.ref 100
.ref 200

* CPW structure
nport:cpw_2 10 20 100 200 filename = "unitcell.yp"

* Transistor small signal model
nport:amplifier 1 2 0 filename = "feedback_ne3210s1.yp"

* Field excitation
gridex:iin 10 100 20 200 ifilename = "unitcell.i"
+ efilename = "dummy.e"

* current meters
vsource:amp1 10 11
vsource:amp2 20 22

* CPW Transmission lines
.model fsa1 cpw (s=.369m w=1m t=10u er=10.8 tand=.001)
cpw:t1 11 100 1 0 model="fsa1" length=8.5m
cpw:t2 22 200 2 0 model="fsa1" length=17.5m

.out write element "vsource:amp1" 0 if in "i1.out"
.out write element "vsource:amp2" 0 if in "i2.out"

* Plot magnitude of current gain
.out plot element "vsource:amp2" 0 if element
+ "vsource:amp1" 0 if div mag db in "igain.out"

* Plot magnitude of voltage gain
.out plot term 20 vf term 10 vf div mag db in "gain.out"

.end
```

Bibliography

[1] M. B. Steer, J. F. Harvey, J. W. Mink, M. N. Abdulla, C. E. Christoffersen, H. M. Gutierrez, P. L. Heron, C. W. Hicks, A. I. Khalil, U. A. Mughal, S. Nakazawa, T. W. Nuteson, J. Patwardhan, S. G. Skaggs, M. A. Summers, S. Wang, and A. B. Yakovlev, "Global modeling of spatially distributed microwave and millimeter-wave systems," *IEEE Trans. Microwave Theory Tech.*, June 1999, pp. 830–839.

[2] C. E. Christoffersen, M. B. Steer and M. A. Summers, "Harmonic balance analysis for systems with circuit-field interactions," *1998 IEEE Int. Microwave Symp. Dig.*, June 1998, pp. 1131–1134.

[3] M. B. Steer, *Transient and Steady-State Analysis of Nonlinear RF and Microwave Circuits*, ECE603 class notes, August 15, 1996.

[4] K. S. Kundert, J. K. White and A. Sangiovanni-Vincentelli, *Steady-state methods for simulating analog and microwave circuits*, Boston, Dordrecht, Kluwer Academic Publishers, 1990.

[5] V. Rizzoli, A. Lipparini, A. Costanzo, F. Mastri, C. Ceccetti, A. Neri and D. Masotti, "State-of-the-Art Harmonic-Balance Simulation of Forced Nonlinear Microwave Circuits by the Piecewise Technique," IEEE Trans. on Microwave Theory and Tech., Vol. 40, No. 1, Jan 1992.

[6] M. Valtonen and T. Veijola, "A microcomputer tool especially suited for microwave circuit design in frequency and time domain," *Proc. URSI/IEEE National Convention on Radio Science*, Espoo, Finland, 1986, p. 20,

[7] M. Valtonen, P. Heikkilä, A. Kankkunen, K. Mannersalo, R. Niutanen, P. Stenius, T. Veijola and J. Virtanen, "APLAC - A new approach to circuit simulation by object orientation," *10th European Conference on Circuit Theory and Design Dig.*, 1991.

[8] K. Mayaram and D. O. Pederson, "CODECS: an object-oriented mixed-level circuit and device simulator," *1987 IEEE Int. Symp. on Circuits and Systems Digest*, 1987, pp 604–607.

[9] A. Davis, "An object-oriented approach to circuit simulation," *1996 IEEE Midwest Symp. on Circuits and Systems Dig.*, 1996, pp 313–316.

[10] B. Melville, P. Feldmann and S. Moinian, "A C++ environment for analog circuit simulation," *1992 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors.*

[11] P. Carvalho, E. Ngoya, J. Rousset and J. Obregon, "Object-oriented design of microwave circuit simulators," *1993 IEEE MTT-S Int. Microwave Symp. Digest*, June 1993, pp 1491–1494.

[12] A. I. Khalil and M. B. Steer "Circuit theory for spatially distributed microwave circuits," *IEEE Trans. on Microwave Theory and Techn.*, Vol. 46, Oct. 1998, pp 1500–1503.

[13] M. Ozkar, *Transient Analysis of Spatially Distributed Microwave Circuits Using Convolution and State Variables*, M. S. Thesis, Department of Electrical and Computer Engineering, North Carolina State University.

[14] A. R. Djordjevic and T. K. Sarkar, "Analysis of time response of lossy multiconductor transmission line networks," *IEEE Trans. on Microwave Theory and Techn.*, Vol. MTT-35, Oct. 1987, pp. 898–908.

[15] D. Winkelstein, R. Pomerleau and M. B. Steer, "Transient simulation of complex, lossy, multi-port transmission line networks with nonlinear digital device termination using a circuit simulator," *Conf. Proc. IEEE SOUTHEASTCON*, Vol. 3, pp. 1239–1244.

[16] T. J. Brazil, "Causal convolution—a new method for the transient analysis of linear systems at microwave frequencies," *IEEE Trans. on Microwave Theory and Techn.*, Vol. 43, Feb. 1995, pp. 315–23.

[17] M. S. Basel, M. B. Steer and P. D. Franzon, "Simulation of high speed interconnects using a convolution-based hierarchical packaging simulator," *IEEE Trans. on Components, Packaging, and Manufacturing Techn.*, Vol. 18, February 1995, pp. 74–82.

[18] J. E. Schutt-Aine and R. Mittra, "Nonlinear transient analysis of coupled transmission lines," *IEEE Trans. on Circuits and Systems*, Vol. 36, Jul. 1989, pp. 959–967.

[19] P. Stenius, P. Heikkilä and M. Valtonen, "Transient analysis of circuits including frequency-dependent components using transgyrator and convolution," *Proc. of the 11th European Conference on Circuit Theory and Design*, Part II, 1993, pp. 1299–1304.

[20] R. Griffith and M. S. Nakhla, "Mixed frequency/time domain analysis of nonlinear circuits," *IEEE Trans. on Computer Aided Design*, Vol.11, Aug. 1992, pp. 1032–43.

[21] P. K. Chan, Comments on "Asymptotic waveform evaluation for timing analysis," *IEEE Trans. on Computer Aided Design*, Vol. 10, Aug. 1991, pp. 1078–79.

[22] M. Celik, O. Ocali, M. A. Tan, and A. Atalar, "Pole-zero computation in microwave circuits using multipoint Padé approximation," *IEEE Trans. on Circuits and Systems*, Jan. 1995, pp. 6–13.

[23] E. Chiprout and M. Nakhla, "Fast nonlinear waveform estimation for large distributed networks," *1992 IEEE MTT-S Int. Microwave Symp. Digest*, Vol.3, Jun. 1992, pp. 1341–1344.

[24] R. J. Trihy and Ronald A. Rohrer, "AWE macromodels for nonlinear circuits," *Proceedings of the 36th Midwest Symposium on Circuits and Systems*, Vol. 1, Aug. 1993, pp. 633–636.

[25] W. T. Beyene and J. E. Schutt-Aineé, "Efficient Transient Simulation of High-Speed Interconnects Characterized by Sampled Data", *IEEE Trans. on Components, Packaging and Manufacturing Techn. — Part B*, Vol. 21, No. 1, Feb. 1998, pp. 105–114.

[26] D. Borisovich and J. E. Schutt-Aineé, "Optimal Transient Simulation of Transmission Lines," *IEEE Trans. on Circuits and Systems—I: Fundamental Theory and Applications*, Vol. 43, No. 2, Feb. 1996, pp. 110–121.

[27] W. Leung and F. Chang, "Transient analysis via fast wavelet-based convolution," *1995 ISCAS Symp. Digest*, Vol. 3, pp. 1884–1887, 1995.

[28] W. Leung and F. Chang, "Wavelet-based waveform relaxation simulation of lossy transmission lines," *1996 ISCAS Symp. Digest*, Vol. 4, pp. 739–742, 1996.

[29] A. Al-Rawi and M. Devaney, "Wavelets and power system transient analysis," *1998 IEEE Instr. and Meas. Techn. Conf. Digest*, Vol. 2 , pp. 1331–1334, 1998.

[30] T. Hisakado and K. Okumura, "Steady states prediction in nonlinear circuit by wavelet transform," *1999 ISCAS Symp. Digest*, 1999.

[31] C. M. Arturi, A. Gandelli, S. Leva, S. Marchi and A. P. Morando, "Multiresolution analysis of time-variant electrical networks," *1999 ISCAS Symp. Digest*, 1999.

[32] W. Cai and J. Wang, "Adaptive multiresolution collocation methods for initial boundary value problems of nonlinear PDEs," *SIAM J. Numer. Anal.*, Vol. 33, No. 3, pp. 937–970, June 1996.

[33] D. Zhou, N. Chen and W. Cai, "A fast wavelet collocation method for high-speed VLSI circuit simulation," *1995 IEEE/ACM ICCAD Symp. Digest*, pp 115–122, 1995.

[34] D. Zhou, X. Li, W. Zhang and W. Cai, "Nonlinear circuit simulation based on adaptive wavelet method," *1997 ISCAS Symp. Digest*, Vol. 3, pp. 1720–1723, 1997.

[35] W. Cai and J. Wang, "An adaptative spline wavelet ADI (SW-ADI) method for two-dimensional reaction-diffusion equations," *J. of Computational Physics*, No. 139, pp. 92–126, 1998.

[36] D. Zhou and W. Cai, "A fast wavelet collocation method for high-speed VLSI circuit simulation," *IEEE Trans. on Circuits and Systems—I: Fundamental Theory and Appl.*, Vol. 46, pp 920–930, Aug. 1999.

[37] D. Zhou, W. Cai and W. Zhang, "An adaptive wavelet method for nonlinear circuit simulation," *IEEE Trans. on Circuits and Systems—I: Fundamental Theory and Appl.*, Vol. 46, pp 931–938, Aug. 1999.

[38] A. Wenzler and E. Lueder, "Analysis of the periodic steady-state in nonlinear circuits using an adaptive function base," *1999 IEEE Int. Symposium on Circuits and Systems Digest.*

[39] R. A. Lippert, T. A. Arias and A. Edelman, "Multiscale computation with interpolating wavelets," *J. of Computational Physics*, No. 140, pp. 278–310, 1998.

[40] A. Graps, "An introduction to wavelets," *IEEE Computational Science and Engineering*, Vol. 2, No. 2, pp. 50–61, Summer 1995.

[41] I. Daubechies, "Ten Lectures on Wavelets," *SIAM Publication*, Philadelphia, 1992.

[42] S. Bertoluzza, "An adaptive collocation method based on interpolating wavelets," *Multiscale Wavelet Methods for Partial Differential Equations*, Academic Press, 1997.

[43] P. Debefve F. Odeh and A. E. Ruehli, "Waveform Techniques" *Circuit Analysis, Simulation and Design*, North-Holland, 1994.

[44] M. S. Nakhla and J. Vlach, "A Piecewise Harmonic Balance Technique for Determination of Periodic Response of Nonlinear Systems," *IEEE Trans. on Circuits and Systems*, Vol CAS-23, No. 2, Feb 1976.

[45] M. B. Steer, C. Chang and G. W. Rhyne, "Computer-Aided Analysis of Nonlinear Microwave Circuits Using Frequency-Domain Nonlinear Analysis Tech.: The State of the Art," *Int. Journal of Microwave and Millimeter-Wave Computer-Aided Engineering*, Vol. 1, No. 2, 181–200, 1991.

[46] N. Borges de Carvalho and J. C. Pedro, "Multitone frequency-domain simulation of nonlinear circuits in large- and small-signal regimes," *IEEE MTT*, Vol. 46, no. 12, pp. 2016–2024, 1998.

[47] V. Borich, J. East and G. Haddad, "An efficient Fourier transform algorithm for multitone harmonic balance," *IEEE Transactions on Microwave Theory and Tech.*, Vol. 47, no. 2, Feb. 1999, pp 182–188.

[48] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold, 1994.

[49] T. J. Brazil, "A new method for the transient simulation of causal linear systems described in the frequency domain," *1992 IEEE MTT-S Int. Microwave Symp. Digest*, June 1992, pp. 1485–1488.

[50] P. Perry and T. J. Brazil, "Hilbert-transform-derived relative group delay," *IEEE Trans. on Microwave Theory and Techn.*, Vol 45, Aug. 1997, pt. 1, pp. 1214–1225.

[51] C. E. Christoffersen, S. Nakazawa, M. A. Summers, and M. B. Steer, "Transient analysis of a spatial power combining amplifier", *1999 IEEE MTT-S Int. Microwave Symp. Dig.*, June 1999, pp. 791–794.

[52] M. Frigo and S. G. Johnson, *FFTW User's Manual*, Massachusetts Institute of Technology, September 1998.

[53] C. Gordon, T. Blazeck and R. Mittra, "Time domain simulation of multiconductor transmission lines with frequency-dependent losses," *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, Vol. 11, Nov. 1992 pp. 1372–87.

[54] M. G. Case, *Nonlinear transmission lines for picosecond pulse, impulse and millimeter-wave harmonic generation*, Ph.D Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara, California, U.S.A., 1993.

[55] M. J. W. Rodwell, M. Kamegawa, R. Yu, M. Case, E. Carman and K. S. Giboney, "GaAs nonlinear transmission lines for picosecond pulse generation and millimeter-wave sampling," *IEEE Trans. on Microwave Theory and Techn.*, Vol. 39, July 1991, pp. 1194–1204.

[56] H. Shi, C. W. Domier and N. C. Luhmann, "A monolithic nonlinear transmission line system for the experimental study of lattice solutions," *J. of Applied Physics*, Vol 4., August 1995, pp. 2558–64.

[57] Compact Software, *Microwave Harmonica Elements Library*, (1994).

[58] A. Brambilla, D. D'Amor and M. Pillan, "Convergence improvements of the harmonic balance method," *Proceedings IEEE Int. Symposium on Circuits and Systems*, Vol. 4 1993, Publ. by IEEE, IEEE Service Center, Piscataway, NJ, USA. p 2482–2485.

[59] R. S. Bain, *NNES user's manual*, 1993.

[60] P. J. C. Rodrigues, *Computer Aided Analysis of Nonlinear Microwave Circuits*, Artech House, 1998.

[61] A. Eliëns, *Principles of object-oriented software development*, Adison-Wesley, 1995.

[62] R. C. Martin. "The dependency inversion principle," *C++ Report*, May 1996.

[63] R. C. Martin, "The Open Closed Principle," *C++ Report*, Jan. 1996.

[64] R. C. Martin, "The Liskov Substitution Principle," *C++ Report*, March 1996.

[65] R. C. Martin, "The Interface Segregation Principle," *C++ Report*, Aug 1996.

[66] R. C. Martin, "UML Tutorial: Part 1 — Class Diagrams," Engineering Notebook Column, *C++ Report*, Aug. 1997.

[67] A. D. Robison, "C++ Gets Faster for Scientific Computing," *Computers in Physics*, Vol. 10, pp. 458–462, 1996.

[68] J. R. Cary and S. G. Shasharina, "Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming," Available from Los Alamos National Laboratory as Report No. LA-UR-96-4064.

[69] The Object Oriented Numerics Page, http://oonumerics.org/.

[70] Silicon Graphics, *Standard Template Library Programmer's Guide*, http://www.sgi.com/Technology/STL/.

[71] T. Veldhuizen, Tech. for Scientific C++ - Version 0.3, Indiana University, Computer Science Department, 1999. (http://extreme.indiana.edu/ tveldhui/papers/Tech./)

[72] A. Griewank, D. Juedes and J. Utke, "Adol-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++," *ACM TOMS*, Vol. 22(2), pp. 131–167, June 1996.

[73] R. Pozo, MV++ v. 1.5a, *Reference Guide*, National Institute of Standards and Technology, 1997.

[74] K. S. Kundert and A. Songiovanni-Vincentelli, *Sparse user's guide - a sparse linear equation solver*, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, Calif. 94720, Version 1.3a, Apr 1988.

[75] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Ph.D. thesis (Under the supervision of W. Gear), Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[76] T. F. Coleman and G. F. Jonsson, "The Efficient Computation of Structured Gradients using Automatic Differentiation," *Cornell Theory Center Technical Report CTC97TR272*, April 28, 1997

[77] S. M. S. Imtiaz and S. M. El-Ghazaly, "Global modeling of millimeter-wave circuits: electromagnetic simulation of amplifiers," *IEEE Trans. on Microwave Theory and Tech.*, vol 45, pp. 2208–2217. Dec. 1997.

[78] C.-N. Kuo, R.-B. Wu, B. Houshmand, and T. Itoh, Modeling of microwave active devices using the FDTD analysis based on the voltage-source approach, *IEEE Microwave Guided Wave Lett.*, Vol. 6, pp. 199–201, May 1996.

[79] E. Larique, S. Mons, D. Baillargeat, S. Verdeyme, M. Aubourg, P. Guillon, and R. Quere, "Electromagnetic analysis for microwave FET modeling," *IEEE microwave and guided wave letters*, Vol 8, pp. 41–43, Jan. 1998.

[80] T. W. Nuteson, H. Hwang, M. B. Steer, K. Naishadham, J.W.Mink, and J. Harvey, "Analysis of finite grid structures with lenses in quasi-optical systems," *IEEE Trans. Microwave Theory Tech.*, pp. 666–672, May 1997.

[81] M. B. Steer, M. N. Abdullah, C. Christoffersen, M. Summers, S. Nakazawa, A. Khalil, and J. Harvey, "Integrated electro-magnetic and circuit modeling of large microwave and millimeter-wave structures," *Proc. 1998 IEEE Antennas and Propagation Symp.*, pp. 478–481, June 1998.

[82] J. Kunisch and I. Wolff, "Steady-state analysis of nonlinear forced and autonomous microwave circuits using the compression approach," *Int. J. of Microwave and Millimeter-Wave Computer-Aided Engineering*, Vol. 5, No. 4, pp. 241–225, 1995

[83] T. H. Cormen, C. E. Leiserson and R. L. Rivest *Introduction to Algorithms*, The MIT Press, McGraw-Hill Book Company, 1990.

[84] Rational Software, UML Resources, *http://www.rational.com/*.

[85] H. S. Tsai, M. J. W. Rodwell and R. A. York, "Planar amplifier array with improved bandwidth using folded-slots," *IEEE Microwave and Guided Wave Letters*, Vol. 4, April 1994, pp. 112–114.

[86] M. B. Steer, M. N. Abdullah, C. Christoffersen, M. Summers, S. Nakazawa, A. Khalil, and J. Harvey, "Integrated electro-magnetic and circuit modeling of large microwave and millimeter-wave structures," *Proc. 1998 IEEE Antennas and Propagation Symp.*, pp. 478–481, June 1998.

[87] M. N. Abdulla, U.A. Mughal, and M B. Steer, "Network Charactarization for a Finite Array of Folded-Slot Antennas for Spatial Power Combining Application," *Proc. 1999 IEEE Antennas and Propagation Symp.*, July 1999.

[88] U. A. Mughal, "Hierarchical approach to global modeling of active antenna arrays," *M.S. Thesis*, North Carolina State University, 1999.

[89] M. A. Summers, C. E. Christoffersen, A. I. Khalil, S. Nakazawa, T. W. Nuteson, M. B. Steer and J. W. Mink, "An integrated electromagnetic and nonlinear circuit simulation environment for spatial power combining systems," *1998 IEEE MTT-S Int. Microwave Symp. Dig.*, June 1998, pp. 1473–1476.

[90] H. Gutierrez, C. E. Christoffersen and M. B. Steer, "An integrated environment for the simulation of electrical, thermal and electromagnetic interactions in high-performance integrated circuits," *Proc. IEEE 6 th Topical Meeting on Electrical Performance of Electronic Packaging*, Sept. 1999.

[91] W. Batty, C. E. Christoffersen, S. David, A. J. Panks, R. G. Johnson, C. M. Snowden and M. B. Steer, "Electro-thermal cad of power devices and circuits with fully physical time-dependent thermal mmodelling of complex 3-d systems," submitted to the *IEEE Trans. on Component and Packaging Technologies*.

[92] W. Batty, C. E. Christoffersen, S. David, A. J. Panks, R. G. Johnson, C. M. Snowden and M. B. Steer, "Fully physical, time-dependent thermal modelling of complex 3-dimensional systems for device and circuit level electro-thermal CAD," submitted to *Semi-Therm XVII*, San Jose, March 2001.

[93] W. Batty, C. E. Christoffersen, S. David, A. J. Panks, R. G. Johnson, C. M. Snowden and M. B. Steer, "Predictive microwave device design by coupled electro-thermal simulation based on a fully physical thermal model," *EDMO 2000*, Glasgow UK, November 2000.

[94] W. Batty, C. E. Christoffersen, S. David, A. J. Panks, R. G. Johnson, C. M. Snowden and M. B. Steer, "Steady-state and transient electro-thermal simulation of power devices and circuits based on a fully physical thermal model," *THERMINIC 2000 Digest*, Budapest, September 2000.

[95] Ptplot. *http://ptolemy.eecs.berkeley.edu/java/ptplot*.

[96] Foty, *MOSFET modeling with SPICE: Principles and Practice*, Prentice Hall, 1997.

[97] Liu, *MOSFET Models for SPICE simulation including BSIM3v3 and BSIM4*, John Wiley and Sons, 2001.

[98] H. Shichman and D. Hodges, "Modeling and Simulation of Insulated-Gate Field-Effect Transistor Switching Circuits," *IEEE J. Sol. St. Circ.* vol. 3, pp. 285-289 (1968)

[99] *Field Effect Transistors*, (ed. by J. Wallmark and H. Johnson), Prentice-Hall, 1966.

[100] Lee, Shur, Fjeldy and Ytterdal, *Semiconductor Device Modeling for VLSI: with the AIM-spice circuit simulator*, Prentice Hall, 1993.

[101] Fjeldy, Ytterdal and Shur, *Introduction to device modeling and circuit simulation*, A Wiley-Interscience Publication, 1998.

[102] Ron M. Kielkowski, *SPICE Practical Device Modeling*, McGraw-Hill, Inc., 1995.

[103] M. B. Steer, "Transient and Steady-State Analysis of Nonlinear RF and Microwave Circuits," ECE 718 class notes, 2001.

[104] C. E. Christoffersen, M. Ozkar, M. B. Steer, M. G. Case and M. Rodwell, "State variable-based transient analysis using convolution," *IEEE Transactions on Microwave Theory and Tech.*, Vol. 47, June 1999, pp. 882–889.

[105] C. E. Christoffersen and M. B. Steer, "State-variable microwave circuit simulation using wavelets," to be published in the *IEEE Microwave and Guided Waves Letters*, 2001.

[106] C. E. Christoffersen, U. A. Mughal and M. B. Steer, "Object Oriented Microwave Circuit Simulation," *Int. J. of RF and Microwave Computer-Aided Engineering*, Vol. 10, Issue 3, 2000, pp. 164–182.

[107] C. E. Christoffersen *Global modeling of nonlinear microwave circuits*, Ph. D. Dissertation, North Carolina State University, December 2000.

[108] C. E. Christoffersen, "Adding Linear Element to Transim", April 2001.

[109] A. Griewank, D. Juedes, J. Utke, "Adol-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++", Version 1.8.2, March 1999.

[110] H. S. Kanj, "Electro-Thermal Resistor Catalog", June 2001.

[111] A.I. Khalil and M.B. Steer, "Circuit theory for spacially distributed microwave circuits", IEEE Trans on Microwave Theory and Technique 46 (1998), 1500-1503.

[112] C. E. Christoffersen and M.B. Steer, "Implementation of the Local Reference Node Concept for Spatially Distributed Circuits", John Willey & Sons, Inc. Int J RF and Microwave CAE 9: 376-384, 1999.