OBJECT-ORIENTED PROGRAMMING IN BIASLISP

by

K. Mayaram

Memorandum No. UCB/ERL M84/103

19 December 1984

(cover)

OBJECT-ORIENTED PROGRAMMING IN BIASLISP

by

Kartikeya Mayaram

Memorandum No. UCB/ERL M84/103

19 December 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Object-Oriented Programming in BIASlisp

ABSTRACT

The LISP-based circuit simulation program BIASlisp now makes use of object-oriented programming through the Flavors system in LISP. This report describes the program structure of this version of BIASlisp. The advantages of object-oriented programming are presented and specific examples are taken from the application at hand to demonstrate the power of such a programming technique. A comparison of perfomance is made with the previous version of BIASlisp and results obtained on the Symbolics 3600 LISP machine and the VAX 11/780 are reported.

ACKNOWLEDGEMENTS

I would like to thank my research advisor Prof. D. O. Pederson for having given me the opportunity to work on this project. His constant support and encouragement have been the driving forces behind this project.

Dr. Andrei Vladimirescu has contributed to this work through numerous discussions. Most of this work was done at Analog Devices, Palo Alto and I would like to thank him for providing an atmosphere that was very conducive to the development of BIASlisp. Thanks to him I have gained more insight into SPICE2.

Ted Vucurevich gave valuable comments and suggestions when I was becoming familiar with the use of flavors. These aided in a better understanding of the Flavors system. His comments on both versions of BIASlisp are also appreciated.

Peter Moore and Rick Spickelmier helped in porting BIASlisp to the VAX 11/780. I thank them for their help and for discussions which were very useful.

Jeff Burns, Ron Gyurscik, and Mark Hofmann have been a big help all through. I wish to thank them for their encouragement and support. Their suggestions and comments are also appreciated.

I thank Jacob White, Res Saleh, Tom Quarles, Mike Klein, and George Jacob for their continuing support.

Last but not least I would like to thank Ann Bikle for help with all administrative matters.

The author thanks the Semiconductor Research Corporation, Analog Devices and Hewlett Packard for funding this research.

1. INTRODUCTION

BIASlisp is a LISP-based circuit simulation program [1-2]. It has been used as a vehicle for studying the performance of LISP-based circuit simulation. The latest version of BIASlisp employs object-oriented programming [3-4] via the **Flavors** system [5]. This report describes the program and makes an evaluation of object-oriented programming for circuit simulation. The specifics of object-oriented programming and the Flavors system on the Symbolics 3600 LISP machine are described in Section 2.

One of the main reasons for using flavors is that object-oriented programming is a more natural way to program, and generic algorithms can be implemented. Modularity is improved and the source code can be easily modified and extended for increased functionality. These ideas are demonstrated by means of an example in Section 4. An important contribution of this work is that these ideas can be used to develop a circuit-simulation program in an object-oriented programming language such as Smalltalk [6-7].

Performance comparison with the previous version of BIASlisp indicates that there is no sacrifice in performance by use of object-oriented programming on the Symbolics 3600 LISP machine. However, on the VAX 11/780 the object-oriented version of BIASlisp has a poor performance. This is due to an increase in the time required for garbage collection. Detailed results and comparisons are presented in Section 5.

2. OBJECT-ORIENTED PROGRAMMING AND FLAVORS

LISP supports object-oriented programming through the Flavors system. This section presents a definition of some of the terminology used in the Flavors system on the LISP machine [5]. The basic ideas are similar to those of any object-oriented language and details of these basics can be found in [3-4].

The fundamental object in the Flavors system is the **flavor**. The flavor can be any conceptual entity and is accompanied by a set of operations that can be performed on it. A flavor basically defines a class of objects that have common characteristics. A program which makes use of flavors can create several instances of a particular flavor or class. Each instance has a set of local variables called **instance variables**. The values of the instance variables is different from instance to instance though their number is the same for all instances of a particular flavor.

Instances of a particular flavor communicate with the "external world" (other parts of the program) by responding to **messages**. A message consists of a symbolic name, the **selector**, which describes the type of operation to be performed by the **receiver** (the object that receives the message). The selector is just the name for a desired task and essentially describes what should happen. Messages have no description of how the task has to be performed. This feature is useful because it hides the details of the implementation. Someone reading a program can concentrate more on understanding the algorithms by knowing what is the result of sending a message instead of being concerned about the specific implementation.

This feature of message passing is very powerful and quite different from a procedure call. A procedure name can be descriptive of the task that it performs. However, there can be only one procedure to a name whereas a message can be interpreted in different ways depending on the type of its receiver. This leads to the idea of generic operations, since the message does not determine what will happen. It is the receiver which responds to a message in a particular way. These ideas are explained further in Sections 3 and 4.

A **method** is a specification of the actions to be performed in responding to a message (similar to a procedure). Associated with each flavor description is the **protocol** of that flavor: a set of methods which define the messages to which instances of the flavor can respond. Methods of a particular flavor can access that flavor's instance variables directly (without sending messages). However, these methods can access instance variables of another flavor only by sending messages. In this sense the instance variables of a flavor are strictly local to the flavor and are also called the **state** of the flavor.

Inheritance is another useful notion in object-oriented programming. A flavor can be made up of other flavors whereby it inherits all the instance variables and methods of the component flavors. The Flavors system makes use of non-hierarchical inheritance [8] which makes it a very powerful system.

Flavors and methods can be defined by use of the functions **defflavor** and **defmethod**, respectively. A message is sent by using the **send** function. The definitions of these functions are included here for completeness and a detailed description is available in [5].

A flavor is defined by the form

(**defflavor** *flavor-name*
((var1 init-val1)
(var2 init-val2) ...
(varn init-valn))
(flav1 flav2 ... flavm)
option1 option2 ...)

Flavor-name is a symbol which is the name of the flavor.

Var1, var2, ..., varn are the names of instance variables and *init-val1, init-val2, ..., init-valn* are their default initial values respectively. An initialization is not required but is useful for assigning default values to the instance variables.

Flav1, flav2, ..., flavm are the names of the component flavors out of which flavor *flavor-name* is built. The features of the component flavors are inherited as described earlier.

Opt1, opt2, etc. are options to the defflavor and are described in [5].

A method is defined by a form

(**defmethod** (*flavor-name message-name) argument-list*
form1 form2 ...)

Flavor-name is a symbol which is the name of the flavor which is to receive the message specified by the symbol *message-name.*

Argument-list is a list of auxiliary variables used by the method (function).

Form1, form2, etc. are the method body.

A message is sent to an object by means of the form

(**send** *object message-name argument-list)*

Object is a flavor instance to which the message *message-name* and the *argument-list* are sent. A message can be handled by a receiver only if the appropriate method has been defined by a **defmethod** otherwise it results in an error.

3. STRUCTURE OF THE PROGRAM

This section describes the choice of objects, the methods and messages used in the BIASlisp circuit-simulation program. The effectiveness of flavor-oriented programming is demonstrated by an example.

The main idea in object-oriented programming is the choice of objects. In a circuit simulation program each type of circuit element can be taken to be a distinct flavor. Thus each allowable circuit element is described by means of a defflavor. The local state of the flavor is specified by its instance variables. Element names, node numbers/names, sparse-matrix pointers, and model parameters are used as the set of instance variables for a particular flavor. Instances of a particular element type are created as they are read in from a circuit description by the input processor of BIASlisp.

The sparse-matrix implementation could also be flavor oriented. However, message sending requires more CPU time and for efficiency reasons structures have been used as described in [2].

The setup and analysis functions send messages to the objects (the instances of a particular element type) for performing specific tasks. If one examines the setup and analysis stages there are two tasks that are object-dependent:

i) the setting up of the sparse-matrix and storing of the direct pointers to appropriate matrix locations as instance variables of each object, and

ii) the calculation and loading of conductances and currents for each element.

If these two tasks can be performed by the object then the setup and analysis stages of the program "know" exactly what has to be done. Therefore, the choice of messages and methods is clear. The objects must be able to create and store their sparse-matrix pointers and calculate and load their contributions into the matrix and right-hand side. The message names that have been used are *:set-matrix-pointers* and *:load-mna-matrix-and-rhs*. Each flavor must have associated methods that describe how these messages have to be handled.

The task of creating the sparse matrix is handled by the function *create-sparse-matrix* and that of loading conductance/currents by the function *load-conductances-and-currents*. To appreciate fully the advantage of message sending the reader is referred to the three subroutines of SPICE2: MATLOC, MATPTR, and LOAD [9]. The function *create-sparse-matrix* performs the same function as subroutines MATLOC and MATPTR and *load-conductances-and-currents* the same as LOAD. However, the implementation is much simpler in terms of details as can be seen from a definition of these two functions for BIASlisp in Figure 1.

The details of how an object sets up the sparse-matrix or how it calculates and loads the conductances are not required by the main program. Hence the program can be developed without worrying about the specifics resulting in generic algorithms which are simpler in description. How the object has to react to a message is specified by the methods of the flavor. This also facilitates addition of new devices and models. Only the appropriate flavors and methods need to be defined or modified to enhance the capabilities of the program. It clearly localizes the portion of the source code that one needs to concentrate on and hence makes it very modular and easy to modify.

Function create-sparse-matrix

```
(defun create-sparse-matrix (dim)
  ;;this function sets up the sparse-matrix pointers and stores the direct
  ;;pointers in the circuit element flavor.
  ;;initialize variables for the sparse-matrix structure

  (setq *col-head* (make-array dim)  ;column header
        *row-head* (make-array dim)  ;row header
        *rhs*      (make-array dim)  ;right hand side vector
        *label-row* (make-array dim) ;to keep track of row swaps
        *label-col* (make-array dim) ;to keep track of column swaps
        *solution* (make-array dim)) ;solution vector
  (fillarray *solution* '(0.0d0))    ;initialize solution vector to 0.0d0
  ;initialize label arrays
  (loop for i from 0 below dim
        do (setf (aref *label-row* i) i)
        (setf (aref *label-col* i) i))

  ;;*element-list* is the list of all circuit elements
  (loop for element in *element-list*
        do
        (send element :set-matrix-pointers)))
```

Function load-conductances-and-currents

```
(defun load-conductances-and-currents ()
  ;;function loads conductances in the nodal admittance matrix and currents
  ;;in the rhs. appropriate methods are called.
  ;;zero rhs and the matrix before each iteration
  (zero-m *number-of-eqns-minus-one*)
  (zero-rhs *number-of-eqns-minus-one*)
  ;;set the element-list to be the list of all resistors, capacitors,
  ;;independent sources and the models that have been defined
  (let ((element-list (append *res-list* *cap-list* *vsrc-list* *isrc-list*
                              *models-defined*)))
    ;cycle through semiconductor device models instead of devices
    (loop for element in element-list
          do (send element :load-mna-matrix-and-rhs)))
  ;;appropriate methods defined for loading the device conductances in the
  ;;mna matrix.
  ;;;load constraints under ".nodeset" option in the input
  (cond (*converge-with-nodeset* (load-constrained-nodes))))
```

Figure 1.
LISP code for Functions create-sparse-matrix and load-conductances-and-currents

4. AN EXAMPLE

In this section a resistor is used as an example of a circuit element and its flavor definition and methods are given. The diode device and model flavors and definitions are explained in Appendix A. Flavors and methods for all circuit elements in BIASlisp are given in Appendix B.

4.1. Flavor definitions

The flavor that is used for the resistor is called *resistor-mixin*. Two basic flavors are defined which specify some common properties of all two-terminal elements. These flavors can then be used as component flavors when defining flavors for other two-terminal elements such as a capacitor or diode.

It is known that a two-terminal element will contribute to four locations in the modified-nodal admittance (MNA) matrix, with the exception of a current source, and the pointers to these must be stored. For this reason the flavor *2-terminal-sparse-matrix-mixin* has been defined. All it does is to provide storage for the four pointers for any two-terminal element. *Since a current source contributes only to the right-hand side this flavor is not a component of the flavor for an independent current source.*

Another set of attributes common to any two-terminal element is the name and node numbers of the element. These common properties are defined by the flavor *2-terminal-element-mixin*. As can be seen in the definition given below, this flavor has the flavor *update-self-mixin* as a component which provides a feature for updating instance variables and is used by all circuit element flavors in this implementation (directly or indirectly). The flavors *2-terminal-sparse-matrix-mixin* and *2-terminal-element-mixin* do not have any methods associated with them. They are only defined to be used as components of other flavors and represent some common features, whereby replication of common information is minimized. Figure 2 gives the definition of these flavors.

```
;define some basic flavors on which a 2-terminal-element flavor can be built
;a 2-terminal-sparse-matrix-mixin is just the set of matrix pointers for a
;two-terminal element.
(defflavor 2-terminal-sparse-matrix-mixin
        (n1n1
         n2n2
         n1n2
         n2n1) ()
  ;;make all variables gettable, settable and initable by use of options.
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

;; another basic flavor for any two-terminal element such as r,c etc.
(defflavor 2-terminal-element-mixin
        (name
         node1
         node2
         value) (update-self-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

Figure 2.
Component Flavors for a resistor-mixin

The two component flavors defined above are now used to define the *resistor-mixin* flavor. The *resistor-mixin* flavor has the *2-terminal-sparse-matrix-mixin* and the *2-terminal-element-mixin* as component flavors directly. By the inheritance feature flavor, *update-self-mixin* is also a component flavor, though indirectly. The set of instance variables for a resistor will be the union of the instance variables of all the component flavors by inheritance. In addition to these instance variables a resistor requires additional variables (specific to a resistor only); the conductance value and the two temperature coefficients. Flavor *resistor-mixin* is given in Figure 3.

4.2. Method definitions

Once the resistor flavor has been defined the associated methods have to be defined. The resistor flavor must be able to handle the messages to set up its sparse-matrix pointers and to load the conductances. Hence two methods are defined.

```
;define the resistor flavor
(defflavor resistor-mixin
      (conductance
       temp-coeff-1
       temp-coeff-2) (2-terminal-sparse-matrix-mixin 2-terminal-element-mixin)
 ;the two flavors defined earlier are now used as component flavors
 :gettable-instance-variables
 :settable-instance-variables
 :initable-instance-variables)
```

Figure 3.
Flavor definition for resistor-mixin

The method which responds to the message *:set-matrix-pointers* calls function *fillij* which creates a matrix entry at the *(i,j)* location indicated by its arguments and returns a pointer to this location. The pointers returned are assigned to the four instance variables *n1n1*, *n2n2*, *n1n2*, and *n2n1* of the resistor flavor (these are present by inheritance of the properties of the *2-terminal-sparse-matrix-mixin* flavor).

For a resistor the message *:load-mna-matrix-and-rhs* must enable the resistor to add its contribution to the admittance matrix. This is the task undertaken by the second method. The conductance value is loaded in the matrix locations to which the resistor contributes. The conductance term adds to the diagonal entries and subtracts from the off-diagonal terms. Function *add-to-matrix-entry (sub-from-matrix-entry)* is used to add (subtract) a value or a list of values to (from) a matrix entry pointed to by *pointer*. The LISP code for these methods is given in Figure 4.

4.3. Remarks

All the flavors and methods that need to be defined for the setup and analysis functions to handle resistors have been defined in Figures 2 to 4. Independent sources require some additional methods and a description is included in Appendix B.

For semiconductor devices the message *load-mna-matrix-and-rhs* is handled by the device models and not by the devices themselves. Each model flavor maintains a list of devices

```
;;method to handle the message :set-matrix-pointers
(defmethod (resistor-mixin :set-matrix-pointers) ()
  ;;create the matrix location and store the pointer.fillij creates a particular
  ;;matrix location and returns the pointer to that location.
  (setq n1n1 (fillij node1 node1 nil)
        n2n2 (fillij node2 node2 nil)
        n1n2 (fillij node1 node2 nil)
        n2n1 (fillij node2 node1 nil)))

;;method to handle the message load-mna-matrix-and-rhs
(defmethod (resistor-mixin :load-mna-matrix-and-rhs)
           ()
  ;;here the conductance value is being added/subtracted to/from
  ;;the previous contents of the appropriate sparse-matrix locations.
  (add-to-matrix-entry n1n1 conductance)
  (add-to-matrix-entry n2n2 conductance)
  (sub-from-matrix-entry n1n2 conductance)
  (sub-from-matrix-entry n2n1 conductance))

;add-to-matrix-entry is a function which takes a pointer to a matrix
;entry and adds "value" to it. sub-from-matrix-entry subtracts "value"
;from the matrix entry pointed to by pointer.
;;          (add-to-matrix-entry pointer value)
;;          (sub-from-matrix-entry pointer value)
```

Figure 4.
Methods for resistor-mixin

which use it as a model. Upon receiving the message *load-mna-matrix-and-rhs* a model flavor instance accesses the model parameters and sends the message *calculate-and-load-conductances* to each of its devices. In this manner the overhead of accessing model parameters is incurred only once for each model. Alternatively the device could handle the message *load-mna-matrix-and-rhs* whereby the model parameters would have to be accessed for each device associated with that model. The first method is efficient since the number of devices in general is more than the number of models.

The addition of a new device or model is fairly easy once the process is understood. One needs to define a flavor for the device or model and the methods which describe the sequence of operations to be performed when a particular message is received. This point is illustrated further in Appendix A by using the diode as an example.

To map into the theme of generic functions it may sometimes be necessary to define methods which do nothing but handle a particular message. An example of this is the independent current source which makes no contribution to the MNA matrix but only to the right-hand side. For the current source then the message *:set-matrix-pointers* really has no meaning. However, it must handle this message otherwise an error will result. So the method is defined and no task is performed. The advantage gained is that the overall setup stage of the program is very general and modular.

5. PERFORMANCE COMPARISON

The performance of the flavor-oriented version of BIASlisp can be appreciated from a comparison with the previous version of the program which does not make use of flavors. The results for the Symbolics 3600 LISP machine are given in Table 1. Both versions of BIASlisp use sparse-matrix techniques and double-precision arithmetic.

Description	Circuit 1		Circuit 2		Circuit 3	
	BIAS1	BIAS2	BIAS1	BIAS2	BIAS1	BIAS2
# iterations	19	17	23	18	43	40
Setup time (sec)	0.5	0.43	0.07	0.06	0.66	0.64
Analysis (sec)	**21.4**	**16.7**	**4.57**	**3.36**	**65.2**	**54.1**
Matrix load (sec)	*15.62*	*12.3*	*3.66*	*2.6*	*57.3*	*47.6*
LU decomp. (sec)	*0.57*	*0.43*	*0.05*	*0.04*	*0.02*	*0.02*
DC solution (sec)	*1.15*	*0.85*	*0.22*	*0.16*	*2.1*	*1.8*
Garbage collection (sec)	0.0	0.0	0.0	0.0	0.0	0.0
Time/iteration (sec)	**1.12**	**0.98**	**0.2**	**0.2**	**1.51**	**1.35**

TABLE 1

BIASlisp Simulation Results on the Symbolics 3600: Comparison of BIAS1 (BIASlisp without flavors) and BIAS2 (BIASlisp with flavors)

It is seen that the flavor implementation results in a smaller number of iterations and is slightly faster. The difference in the number of iterations may be due to a changed order in loading the MNA matrix. Complete runtime statistics do not give a meaningful comparison when the number of iterations are significantly different. The only reasonable parameter for comparison is the time per iteration. However, the complete statistics have been included to give an idea where the program spends most of its time. Most of the time is spent in the matrix load phase. Garbage collection was turned off during these simulations, therefore, no time is spent in garbage collection

The flavor implementation is efficient and is a good choice for such an application. Message passing does have an overhead but that has not resulted in a degraded overall performance on the Symbolics 3600. Simulation performance on a recent release of the Symbolics

3600 system software indicates a significant improvement in speed for the flavor implementation of BIASlisp [10]. These results are presented in Table 2.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	19	18	40
Setup time (sec)	0.55	0.06	0.51
Analysis (sec)	**14.16**	**3.04**	**51.8**
Matrix load (sec)	*10.64*	*2.36*	*43.5*
LU decomp. (sec)	*1.17*	*0.07*	*0.83*
DC solution (sec)	*1.00*	*0.16*	*1.47*
Time/iteration (sec)	**0.69**	**0.15**	**1.17**

TABLE 2

BIASlisp Simulation Results on the Symbolics 3600: Under software version 6.0 using the flavor implementation

The results for the VAX 11/780 are included in Table 3. Here again the total number of iterations is different for the two versions and the time/iteration is used as the basis for comparison. It is seen that the garbage collection time takes up almost fifty percent of the total analysis time. If one subtracts the garbage collection time in both cases then the time/iteration works out to be almost the same as indicated in Table 4. However, the garbage collection time cannot be ignored in a real application and does result in a performance penalty as has been pointed out in [2]. It can be speeded up but it cannot be avoided.

Description	Circuit 1		Circuit 2		Circuit 3	
	BIAS1	BIAS2	BIAS1	BIAS2	BIAS1	BIAS2
# iterations	14	22	23	18	26	40
Setup time (sec)	7.2	13.25	1.67	1.2	9.9	10.1
Analysis (sec)	**53.35**	**113.8**	**23.05**	**24.55**	**136.2**	**290.8**
Matrix load (sec)	*45.1*	*87.1*	*17.3*	*21.2*	*123.6*	*266.4*
LU decomp. (sec)	*1.17*	*13.5*	*0.3*	*0.2*	*0.4*	*2.05*
DC solution (sec)	*4.32*	*10.4*	*2.5*	*2.0*	*7.1*	*16.75*
Garbage collection (sec)	16.8	54.3	7.4	13.45	40.5	133.4
Time/iteration (sec)	**3.81**	**5.17**	**1.0**	**1.36**	**5.24**	**7.27**

TABLE 3

**BIASlisp Simulation Results on the VAX 11/780:
Comparison of BIAS1 (BIASlisp without flavors)
and BIAS2 (BIASlisp with flavors)**

Description	Circuit 1	Circuit 2	Circuit 3
Time/iteration (sec) (BIASlisp without flavors)	2.6	0.64	3.7
time/iteration (sec) (BIASlisp using flavors)	2.7	0.68	3.94

TABLE 4

**Time/iteration comparison ignoring garbage collection time
on the VAX 11/780**

6. CONCLUSION

An object-oriented version of the circuit simulation program BIASlisp has been developed. This program has been used to motivate the usefulness of object-oriented programming for circuit simulation.

This program can be easily used with other kinds of design tools which also employ the object-oriented paradigm. As an example consider a schematic-capture program. This program can represent each circuit element as a flavor and have methods for doing schematics editing. By using the flavors, the methods and the functions of BIASlisp, the schematic capture program can be extended to provide simulation capabilities as well. Modularity is an important consequence of the flavor system and is well suited for developing very complex programs. The code is easy to comprehend because generic algorithms can be described.

Since the flavor system is LISP-based one still enjoys the advantages of the LISP programming environment. This, along with the power of object-oriented programming, provides an extremely productive system for software development.

Appendix A

This appendix explains the flavors and methods for a diode device and model and illustrates how new models can be added to BIASlisp

A1. Diode Model Flavors

The flavors that are used by the diode model are defined in this section. In Section A1.1 the component flavors for any semiconductor device model are presented.

A1.1. Component flavors

Each device model has a model name and a list of devices which make use of this model. As has been noted in Section 4 the conductances for a semiconductor device are calculated by cycling through models rather than cycling through devices (as in SPICE2). When a particular model is referred to, all its parameters are accessed. These parameters are common to all devices which make use of the model. Hence, this accessing is done only once for all devices of that model. If a device is referred to, then the overhead of accessing parameters for the model associated with the device is incurred every time conductance/current calculations are made for the device. Clearly, the first method is efficient and has been used in the flavor-oriented implementation of BIASlisp.

The flavor *model-mixin* is used to represent information common to all models. Only a name and a device-list are characteristics common to different types of models and these are the instance variables of *model-mixin*. Flavor *update-self-mixin* is used as a component flavor.

Each model creates a list of devices that are associated with it by means of the message *:update-device-list*. A device name is passed as a parameter to the method which handles this message and the device name gets added to the list of devices already existent. Figure A1 gives the LISP code for these flavors and methods.

A function for calculating model constants is defined by *calculate-model-constants*. This function sends a message to every model that has been defined, to compute values for constants which are to be used in calculating conductances and currents. Since these calculations are done prior to analysis, there is a saving of time in the analysis phase of the program. A description for this function is also included in Figure A1.

```
;;flavors for a generic semiconductor device model
(defflavor model-mixin ((name nil)
                        (device-list nil)) (update-self-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)


;;a method to create the list of devices which are associated
;;with a model. this is used before the setup phase.
(defmethod (model-mixin :update-device-list) (device)
  (setq device-list (cons device device-list)))

;;a function to calculate some commonly used model constants.
(defun calculate-model-constants ()
  (loop for model in *models-defined*
        do (send model :calculate-model-constants)))
```

Figure A1.
Component Flavors and Methods

A1.2. The Model Flavors

A description of the diode model flavors and methods is given. The model used is a dc one and the model parameter is the saturation current *is*. No breakdown characteristics have been modeled.

The diode model is defined by the flavor *diode-model-mixin*. Its instance variables are the model type, **type*, the saturation current, **is*, and the critical voltage used for limiting purposes, **vcrit*. *Model-mixin* defined previously is used as a component flavor. Two methods are defined to handle the messages *:calculate-model-constants* and *:load-mna-matrix-and-rhs*. In this example **vcrit* is the only model constant that is calculated. The method for the

message *:load-mna-matrix-and-rhs* cycles through the device-list and sends a message *:calculate-and-load-conductances* to each device. When an instance of the device receives this message, it performs the task requested. These flavors and methods are given in Figure A2.

```
;;flavors used for diode models.

(defflavor diode-model-mixin ((*type 'diode)
                              (*is 1d-14)
                              (*vcrit nil)) (model-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)


(defmethod (diode-model-mixin :calculate-model-constants) ()
  (setq *vcrit (* *vt* (log (// *vt* 1.4142 *is)))))

;; method :load-mna-matrix is defined for the model for reasons
;; of efficiency. the program cycles through models instead of
;; devices (SPICE cycles through devices) and hence
;; this method is associated with the model flavor.
(defmethod (diode-model-mixin :load-mna-matrix) ()
  (let* ((is *is)
         (vcrit *vcrit))
    (loop for device in device-list
          do (send device :calculate-and-load-conductances))))
```

Figure A2.
Diode Model Flavors and Methods

A2. Diode Device Flavors and methods

This section gives a description of the flavors and methods that have been used for a diode device.

The flavor for a diode device is denoted by *diode-device-mixin*. Instance variables are the name of the model associated with the device, *model-name*, the area factor, *area*, and the previous values of the voltage across the device, the current through it, and the conductance denoted by *vdo*, *ido*, and *gdo* respectively. Flavors *2-terminal-element-mixin* and *2-terminal-sparse-matrix-mixin* are used as component flavors (as in the case of a resistor flavor). The message *:set-matrix-pointers* is handled in the same way as it was for a resistor in Section 4.

The diode flavor and the method for *:set-matrix-pointers* are given in Figure A3.

```
;;define diode flavors and associated methods
(defflavor diode-device-mixin
      (model-name area vdo ido gdo)
      (2-terminal-element-mixin 2-terminal-sparse-matrix-mixin)
 :gettable-instance-variables
 :settable-instance-variables
 :initable-instance-variables)

;;set-up the sparse-matrix pointers for the diode device
(defmethod (diode-device-mixin :set-matrix-pointers) ()
 (setq n1n1 (fillij node1 node1 nil)
      n2n2 (fillij node2 node2 nil)
      n1n2 (fillij node1 node2 nil)
      n2n1 (fillij node2 node1 nil)))
```

Figure A3.
Diode Flavor and Method for :set-matrix-pointers

The method for handling the message *:calculate-conductances-and-currents* is given in Figure A4.This message is sent by the model associated with the device. An outline of the operations performed in this method are as follows: If it is the first iteration then a guess is used for the diode voltage, otherwise the voltage is calculated as a difference of the two node voltages from the previous iteration. This junction voltage is then limited by the function *pnjlim* and stored. A call to the function *calculate-conductances-and-currents* does all the conductance and current calculations. A special type of definition for this function enables it to use the instance variables of the flavor (such a function is defined by *defun-method*). After the conductance and current terms have been calculated a check is made for convergence and then the contribution to the matrix and right-hand side is entered. The functions *calculate-conductances-and-currents*, *check-convergence*, and *load-currents-and-conductances* are given in Figure A5.

```
;;defun methods for calculating diode conductances etc...
(defmethod (diode-device-mixin :calculate-and-load-conductances) ()
 (let ((isat (* area is))
       vd
       delvd
       idhat)
  (cond ((and (= 1 *iteration-number*) *use-initial-guess*)
         (setq vdo vcrit))
        (t (setq vd (- (aref *old-sol* node1) (aref *old-sol* node2))
                 delvd (- vd vdo)
                 idhat (+ ido (* gdo delvd))
                 vdo (pnjlim vd vdo vcrit))))
  ;;compute branch currents and derivatives
  (calculate-conductances-and-currents isat)
  ;;check convergence
  (check-convergence idhat)
  ;;load rhs and yn-matrix
  (load-currents-and-conductances)))
```

Figure A4.
Method for calculation and loading of the current and conductance for a diode

A3. Remarks

A simple dc model for a diode has been used as an example to demonstrate the power of flavor-oriented programming. It is easy to extend the model to include transient, and reverse breakdown behavior. The method that handles the message *:load-conductances-and-currents* for the device has to be modified to include these capabilities. This is all that need be done to extend the above model. The above example illustrates the ease with which different device models can be implemented. A complete dc and transient model for a MOSFET is given in Appendix B.

```
(defun-method calculate-conductances-and-currents diode-device-mixin
  (isat)
  (let (exp-vd)
    (cond ((<= vdo (* -5.0 *vt*)) (setq gdo (+ (// isat *vt*) *gmin*)
                                      ido (* vdo (- gdo *gmin*))))
          (t (setq exp-vd (exp (// vdo *vt*))
                   gdo (+ (// (* isat exp-vd) *vt*) *gmin*)
                   ido (- (* *vt* (- gdo *gmin*)) isat))))))


(defun-method check-convergence diode-device-mixin (idhat)
  (let (tolerance)
    (cond ((not (= 1 *iteration-number*))
           (setq tolerance (+ *abstol* (* *reltol* (max (abs idhat) (abs ido)))))
           (cond ((>= (abs (- idhat ido)) tolerance)
                  (setq *number-not-converged* (1+ *number-not-converged*)))))
          (t (setq *number-not-converged* (1+ *number-not-converged*))))))


(defun-method load-currents-and-conductances diode-device-mixin ()
  (let ((ideq (- ido (* gdo vdo))))
    (sub-from-rhs node1 ideq)
    (add-to-rhs   node2 ideq)
    (add-to-matrix-entry   n1n1 gdo)
    (add-to-matrix-entry   n2n2 gdo)
    (sub-from-matrix-entry n1n2 gdo)
    (sub-from-matrix-entry n2n1 gdo)))
```

Figure A5.
Functions used by the diode methods

APPENDIX B

This appendix describes all the flavors and methods that have been used in the flavor-oriented implementation of BIASlisp.

USEFUL COMPONENT FLAVORS

```
;;some other useful flavor mixins used by biaslm

;;flavor update-self-mixin used by any flavor instance to update it's instance variables
;;a useful function when parsing input information
```

update-self-mixin

```
(defflavor update-self-mixin () ()
  (:method-combination (:append :base-flavor-first :update-self)))
```

update-self-mixin

```
(defmethod (update-self-mixin :update-self) (keywords)
  (let* ((result nil) (keyword nil))
    (do ((fl1 (flavor-depends-on-all (instance-flavor self))
              (cdr fl1)))
        ((null fl1) result)
      (do ((l (flavor-initable-instance-variables (get (car fl1) 'flavor))
              (cdr l)))
          ((null l))
        (setq keyword (member (caar l) keywords))
        (unless (null keyword)
          (send self (keyword-hack (first keyword)) (eval (second keyword))))))))
```

keyword-hack

```
(defun keyword-hack (keyword)
  `(quote ,(uconcat ":set-" keyword)))
```

```
;;a two-terminal sparse-matrix mixin is defined as a base flavor whish is used by r,c,v,i, and d
```

2-terminal-sparse-matrix-mixin

```
(defflavor 2-terminal-sparse-matrix-mixin
        (n1n1
         n2n2
         n1n2
         n2n1) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

;;;presently structures have been used for the sparse-matrix implementation
```

sparse-elem

...sparse-elem

```
(defstruct (sparse-elem)
  sparse-elem-value
  sparse-elem-i
  sparse-elem-j
  sparse-elem-nexti
  sparse-elem-nextj)

;; a basic flavor for any two terminal element such as r,c etc.
```

2-terminal-element-mixin

```
(defflavor 2-terminal-element-mixin
        (name node1 node2 value) (update-self-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

;a useful macro for entering data in the matrix.
```

add-to-matrix-entry

```
(defmacro add-to-matrix-entry (pointer &rest value)
 `(setf (sparse-elem-value ,pointer) (+ (sparse-elem-value ,pointer) ,@value)))
```

sub-from-matrix-entry

```
(defmacro sub-from-matrix-entry (pointer &rest value)
 `(setf (sparse-elem-value ,pointer) (- (sparse-elem-value ,pointer) ,@value)))
```

add-to-rhs

```
(defmacro add-to-rhs (node &rest value)
 `(setf (aref *rhs* ,node) (+ (aref *rhs* ,node) ,@value)))
```

sub-from-rhs

```
(defmacro sub-from-rhs (node &rest value)
 `(setf (aref *rhs* ,node) (- (aref *rhs* ,node) ,@value)))
;;FLAVORS FOR A GENERIC SEMICONDUCTOR DEVICE MODEL
```

model-mixin

```
(defflavor model-mixin ((name nil)
                        (device-list nil)) (update-self-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

model-mixin

```
(defmethod (model-mixin :update-device-list) (device)
  (setq device-list (cons device device-list)))
```

calculate-model-constants

```
(defun calculate-model-constants ()
  (loop for model in *models-defined*
        do (send model ':calculate-model-constants)))
```

...calculate-model-constants

```
;;FLAVOR STATE-INFO-MIXIN IS USED BY CAPACITIVE ELEMENTS TO STORE THE
CHARGE AND CURRENT STATE
;;FOR USE AT THE NEXT TIME POINT. CAN BE EXTENDED IF TRUNCATION ERROR IS
USED FOR TIMESTEP CONTROL
```

state-info-mixin

```
(defflavor state-info-mixin
        ;;used to store charge and current info of capacitive elements
        ((q-present-state (make-hash-table)) (q-previous-state (make-hash-table))
         (i-present-state (make-hash-table)) (i-previous-state (make-hash-table))) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

state-info-mixin

```
(defmethod (state-info-mixin :update-state-info) ()
  ;;updates the state table after a timepoint has been accepted :the tables are cycled over
  (let (temp)
  (setq temp q-previous-state
        q-previous-state q-present-state
        q-present-state temp
        temp i-previous-state
        i-previous-state i-present-state
        i-present-state temp)))
```

update-all-state-info

```
(defun update-all-state-info ()
  (let ((element-list (append *cap-list* *mos-list*)))
    (loop for element in element-list
          do (send element ':update-state-info))))
```

store-state-info

```
(defun store-state-info (state-table key value)
    (puthash key value state-table))
```

get-state-info

```
(defun get-state-info (state-table key)
  (gethash key state-table))
```

RESISTOR FLAVORS AND METHODS

resistor-mixin

```
(defflavor resistor-mixin
        (conductance temp-coeff-1 temp-coeff-2) (2-terminal-sparse-matrix-mixin 2-terminal-element-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

resistor-mixin

```
(defmethod (resistor-mixin :set-matrix-pointers) ()
  (setq n1n1 (fillij node1 node1 nil)
        n2n2 (fillij node2 node2 nil)
        n1n2 (fillij node1 node2 nil)
        n2n1 (fillij node2 node1 nil)))
```

resistor-mixin

```
(defmethod (resistor-mixin :load-mna-matrix-and-rhs)
           ()
  (add-to-matrix-entry  n1n1 conductance)
  (add-to-matrix-entry  n2n2 conductance)
  (sub-from-matrix-entry n1n2 conductance)
  (sub-from-matrix-entry n2n1 conductance))
```

CAPACITOR FLAVORS AND METHODS

capacitor-mixin

```
(defflavor capacitor-mixin
        (initial-condition)
        (2-terminal-sparse-matrix-mixin 2-terminal-element-mixin state-info-mixin)
;;the q-state-table stores information about charges and the i-state-table stores information about currents
      :gettable-instance-variables
      :settable-instance-variables
      :initable-instance-variables)
```

capacitor-mixin

```
(defmethod (capacitor-mixin :set-matrix-pointers) ()
  (setq n1n1 (fillij node1 node1 nil)
        n2n2 (fillij node2 node2 nil)
        n1n2 (fillij node1 node2 nil)
        n2n1 (fillij node2 node1 nil)))
```

calculate-capacitor-charge

```
(defun calculate-capacitor-charge (name n1 n2 value vc q-previous-state q-present-state)
  (let (qc)
  (cond (*new-time*
         (cond ((and (= 1 *number-timepoints*) (= 1 *iteration-number*))
                (cond (*use-initial-conditions*
                       (cond ((not vc) (setq vc 0.0d0))))
                      (t (setq vc (- (aref *old-sol* n1) (aref *old-sol* n2)))))
                ;;use the dc solution to get voltages
                (setq qc (* vc value))
                (store-state-info q-previous-state name qc))
               (t (setq qc (get-state-info q-previous-state name)))))
         (t (setq vc (- (aref *old-sol* n1) (aref *old-sol* n2))
                qc (* vc value))))
  (store-state-info q-present-state name qc)))
```

get-capacitor-contribution

```
(defun get-capacitor-contribution (name value q-previous-state q-present-state i-previous-state i-present-state)
    ;;this function performs the numerical integration for a capacitor . trapezoidal integration with
    a backward-euler
    ;;start up has been used. states are assumed to be stored in the state tables for charge and current.
    (prog (ieq geq qc qco ico)
          (setq qc  (get-state-info q-present-state name)
                qco (get-state-info q-previous-state name)
                ico (get-state-info i-previous-state name))
    ;;check if first timepoint
          (cond ((= 1 *number-timepoints*)  ;;use backward-euler integration
                 (store-state-info i-present-state name (// (- qc qco) *delta*))  ;;i(n+1)
                 (setq geq (// value *delta*)  ;;equivalent conductance
                       ieq (- (// qco *delta*)));;equivalent current only valid for a linear capacitor
```

...get-capacitor-contribution

```
    (return (list geq ieq)))
   (t                          ;use trapezoidal integration
    (store-state-info i-present-state name (- (* 2 (// (- qc qco) *delta*)) ico))
     (setq geq (// (* 2 value) *delta*)
           ieq (- (+ ico (// (* 2 qco) *delta*))))
     (return (list geq ieq))))))
```

capacitor-mixin

```
(defmethod (capacitor-mixin :load-mna-matrix-and-rhs) ()
  (cond ((and *transient-analysis* (not *use-initial-guess*))
         ;;capacitance contribution to the mna matrix during transient analysis only
         (calculate-capacitor-charge name node1 node2 value initial-condition
                                     q-previous-state q-present-state)
         (destructuring-bind (equivalent-conductance equivalent-current)
                             (get-capacitor-contribution name value
                                                         q-previous-state q-present-state
                                                         i-previous-state i-present-state)
         ;;load the equivalent current and conductance

         (sub-from-rhs node1 equivalent-current)
         (add-to-rhs   node2 equivalent-current)
         (add-to-matrix-entry   n1n1 equivalent-conductance)
         (add-to-matrix-entry   n2n2 equivalent-conductance)
         (sub-from-matrix-entry n1n2 equivalent-conductance)
         (sub-from-matrix-entry n2n1 equivalent-conductance)))))
```

SOURCE FLAVORS AND METHODS

dc-mixin

```
(defflavor dc-mixin (dc-value) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pwl-mixin

```
(defflavor pwl-mixin
        (time-voltage-data-list dc-value) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

sin-mixin

```
(defflavor sin-mixin
        (v1 v2 omega delay theta dc-value) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pulse-mixin

```
(defflavor pulse-mixin
        (v1 v2 td tr pw tf per v2-minus-v1 td+tr td+tr+pw td+tr+pw+tf td+per dc-value) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

independent-vsrc-mixin

```
(defflavor independent-vsrc-mixin
        (branch-number dc-value (ac-magnitude 0.0d0) (ac-phase 0.0d0))
      (2-terminal-element-mixin 2-terminal-sparse-matrix-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

independent-vsrc-mixin

```
(defmethod (independent-vsrc-mixin :set-matrix-pointers) ()
  (setq n1n1 (fillij node1 branch-number nil)
        n2n2 (fillij branch-number node1 nil)
        n1n2 (fillij node2 branch-number nil)
        n2n1 (fillij branch-number node2 nil)))
```

independent-vsrc-mixin

```
(defmethod (independent-vsrc-mixin :interchange-rows) ()
  (cond ((not (zerop node1)) (swaprow node1 branch-number))
        ((not (zerop node2)) (swaprow node2 branch-number))))
```

independent-vsrc-mixin

```
(defmethod (independent-vsrc-mixin :load-mna-matrix-and-rhs) ()
  (add-to-matrix-entry   n1n1 1.0d0)
  (add-to-matrix-entry   n2n2 1.0d0)
  (sub-from-matrix-entry n1n2 1.0d0)
  (sub-from-matrix-entry n2n1 1.0d0)
  (setf (aref *rhs* branch-number) dc-value))
```

independent-isrc-mixin

```
(defflavor independent-isrc-mixin
        (dc-value (ac-magnitude 0.0d0) (ac-phase 0.0d0)) (2-terminal-element-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

independent-isrc-mixin

```
(defmethod (independent-isrc-mixin :set-matrix-pointers) ())
```

independent-isrc-mixin

```
(defmethod (independent-isrc-mixin :load-mna-matrix-and-rhs) ()
  (sub-from-rhs node1 dc-value)
  (add-to-rhs node2   dc-value))
```

dc-vsrc-mixin

```
(defflavor dc-vsrc-mixin ()
          (independent-vsrc-mixin dc-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pulse-vsrc-mixin

```
(defflavor pulse-vsrc-mixin ()
          (independent-vsrc-mixin pulse-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

sin-vsrc-mixin

```
(defflavor sin-vsrc-mixin ()
          (independent-vsrc-mixin sin-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pwl-vsrc-mixin

```
(defflavor pwl-vsrc-mixin ()
          (independent-vsrc-mixin pwl-mixin)
  :gettable-instance-variables
  :settable-instance-variables
```

...pwl-vsrc-mixin

```
  :initable-instance-variables)
```

dc-isrc-mixin

```
(defflavor dc-isrc-mixin ()
        (independent-isrc-mixin dc-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pulse-isrc-mixin

```
(defflavor pulse-isrc-mixin ()
        (independent-isrc-mixin pulse-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

sin-isrc-mixin

```
(defflavor sin-isrc-mixin ()
        (independent-isrc-mixin sin-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

pwl-isrc-mixin

```
(defflavor pwl-isrc-mixin ()
        (independent-isrc-mixin pwl-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

;; A FLAVOR IMPLEMENTATION FOR EACH TYPE OF SOURCE (PULSE, SIN, AND PWL)

dc-mixin

```
(defmethod (dc-mixin :update-value) ())
```

pulse-mixin

```
(defmethod (pulse-mixin :update-value) ()
  (let ((time *present-time*))
    (setq time (loop for timei first time then (- timei per)
                     until (<= timei td+per)
                     finally (return timei)))
    (setq dc-value (cond ((<= time td) v1)
                         ((<= time td+tr) (+ v1 (*(// (- time td) tr) v2-minus-v1)))
                         ((<= time td+tr+pw) v2)
                         ((<= time td+tr+pw+tf) (+ v2 (*(// (- time td+tr+pw) tf) (- v2-minus-v1))))
                         (t v1)))))
```

sin-mixin

...sin-mixin

```
(defmethod (sin-mixin :update-value) ()
  (let ((time (- *present-time* delay)))
     (setq dc-value (cond ((<= time 0.0d0) v1)
          (t
           (cond ((zerop theta)
                  (+ v1 (*v2 (sin (*omega time)))))
                 (t (+ v1 (*v2 (sin (*omega time)) (exp (- (*time theta))))))))))))
```

pwl-mixin

```
(defmethod (pwl-mixin :update-value) ()
  (let ((t1 (first time-voltage-data-list))
        (v1 (second time-voltage-data-list)))
    (loop for list first (rest2 time-voltage-data-list) then (rest2 list)
          while list
          for t2 = (first list)
          for v2 = (second list)
          when (> *present-time* t2)
          do (setq t1 t2 v1 v2)
          else
          do (setq dc-value (+ v1 (* (// (- *present-time* t1) (- t2 t1)) (- v2 v1))))
          (return))))
```

;; FUNCTION TO UPDATE SOURCE VALUES DURING TRANSIENT ANALYSIS

update-sources

```
(defun update-sources ()
  (let ((source-list (append *vsrc-list* *isrc-list*)))
       (loop for source in source-list
             do (send source ':update-value))))
```

;;to implement a break-point table for transient analysis define appropriate
;;methods (defun-method?)

dc-mixin

```
(defmethod (dc-mixin :generate-break-points) ())
```

pulse-mixin

```
(defmethod (pulse-mixin :generate-break-points) ()
  (loop for time first 0.0d0 then (+ time per)
        while (< time *tstop*)
        do
        (nconc *break-point-table* (list (+ time td)
                                         (+ time td+tr)
                                         (+ time td+tr+pw)
                                         (+ time td+tr+pw+tf)))))
```

sin-mixin

```
(defmethod (sin-mixin :generate-break-points) ()
  (nconc *break-point-table* (list delay)))
```

...sin-mixin

pwl-mixin

```
(defmethod (pwl-mixin :generate-break-points) ()
  (loop for data-list first time-voltage-data-list then (rest2 data-list)
        for time = (first data-list)
        while (and time (< time *tstop*))
        do
        (nconc *break-point-table* (list time))))
```

DIODE FLAVORS AND METHODS

;;define diode flavors and associated methods

diode-device-mixin

```
(defflavor diode-device-mixin
        (model-name area initial-condition vdo ido gdo) (2-terminal-element-mixin 2-terminal-sparse-matrix)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

;;set-up the sparse-matrix pointers for the diode device

diode-device-mixin

```
(defmethod (diode-device-mixin :set-matrix-pointers) ()
  (setq n1n1 (fillij node1 node1 nil)
        n2n2 (fillij node2 node2 nil)
        n1n2 (fillij node1 node2 nil)
        n2n1 (fillij node2 node1 nil)))
```

;;defun methods for calculating diode conductances etc...

```
(declare (special is vcrit))
```

calculate-conductances-and-currents

```
(defun-method calculate-conductances-and-currents diode-device-mixin (isat)
  (let (exp-vd)
    (cond ((<= vdo (* -5.0 *vt*)) (setq gdo (+ (// isat *vt*) *gmin*)
                                          ido (* vdo (- gdo *gmin*))))
          (t (setq exp-vd (exp (// vdo *vt*))
                   gdo (+ (// (* isat exp-vd) *vt*) *gmin*)
                   ido (- (* *vt* (- gdo *gmin*)) isat))))))
```

check-convergence

```
(defun-method check-convergence diode-device-mixin (idhat)
  (let (tolerance)
    (cond ((not (= 1 *iteration-number*))
           (setq tolerance (+ *abstol* (* *reltol* (max (abs idhat) (abs ido))))))
(cond ((>= (abs (- idhat ido)) tolerance) (setq *number-not-converged* (1+ *number-not-converged*))))
          (t (setq *number-not-converged* (1+ *number-not-converged*))))))
```

load-currents-and-conductances

```
(defun-method load-currents-and-conductances diode-device-mixin ()
  (let ((ideq (- ido (* gdo vdo))))
    (sub-from-rhs node1 ideq)
    (add-to-rhs node2  ideq)
    (add-to-matrix-entry  n1n1 gdo)
    (add-to-matrix-entry  n2n2 gdo)
    (sub-from-matrix-entry n1n2 gdo)
    (sub-from-matrix-entry n2n1 gdo)))
```

...load-currents-and-conductances

diode-device-mixin

```
(defmethod (diode-device-mixin :calculate-and-load-conductances) ()
  (let ((isat (* area is))
        vd
        delvd
        idhat)
    (cond ((and (= 1 *iteration-number*) *use-initial-guess*)
           (setq vdo vcrit))
          (t (setq vd (- (aref *old-sol* node1) (aref *old-sol* node2))
                   delvd (- vd vdo)
                   idhat (+ ido (* gdo delvd))
                   vdo (pnjlim vd vdo vcrit))))
    ;;compute branch currents and derivatives
    (calculate-conductances-and-currents isat)
    ;;check convergence
    (check-convergence idhat)
    ;;load rhs and yn-matrix
    (load-currents-and-conductances)))
```

;; flavors used for diode models.

diode-model-mixin

```
(defflavor diode-model-mixin ((*type 'diode)
                              (*is 1d-14)
                              (*vcrit nil)) (model-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

diode-model-mixin

```
(defmethod (diode-model-mixin :calculate-model-constants) ()
  (setq *vcrit (* *vt* (log (// *vt* 1.4142 *is)))))
```

;; method :load-mna-matrix-and-rhs is defined for the model for reasons of efficiency. the program
;; cycles through models instead of devices (SPICE cycles through devices) and hence
;; this method is associated with the model flavor.

diode-model-mixin

```
(defmethod (diode-model-mixin :load-mna-matrix-and-rhs) ()
  (let* ((is *is)
         (vcrit *vcrit))
    (loop for device in device-list
          do (send device ':calculate-and-load-conductances))))
```

BIPOLAR FLAVORS AND METHODS

;define bipolar flavors and associated methods
;Eber's-Moll dc model is used at present

bjt-device-sparse-matrix-mixin

```
(defflavor bjt-device-sparse-matrix-mixin
  (ncnc ncnb ncne nbnc nbnb nbne nenc nenb nene) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

bjt-device-mixin

```
(defflavor bjt-device-mixin
  (name nc nb ne model-name area
        vbeo vbco ico ibo gibvbe gibvbc gicvbe gicvbc)
  (bjt-device-sparse-matrix-mixin update-self-mixin state-info-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

bjt-device-mixin

```
(defmethod (bjt-device-mixin :set-matrix-pointers) ()
  (setq ncnc (fillij nc nc nil)
        ncnb (fillij nc nb nil)
        ncne (fillij nc ne nil)
        nbnc (fillij nb nc nil)
        nbnb (fillij nb nb nil)
        nbne (fillij nb ne nil)
        nenc (fillij ne nc nil)
        nenb (fillij ne nb nil)
        nene (fillij ne ne nil)))
```

;declare special all model parameters

```
(declare (special vaf betaf betar is vcrit type))
```

calculate-conductances-and-currents

```
(defun-method calculate-conductances-and-currents bjt-device-mixin (isat)
  (let* ((exp-vbe (exp (f// vbeo *vt*)))
         (exp-vbc (exp (f// vbco *vt*)))
         (idume (f* exp-vbe isat))
         (idumc (f* exp-vbc isat))
         (1/alphar (f+ 1.0d0 (f// 1.0d0 betar)))
         (vabc    (f- 1.0d0 (f// vbco vaf))))
        (setq gicvbe (f// (f* idume vabc) *vt*)
              gibvbe (f// gicvbe betaf)
              gibvbc (f// (f* idumc vabc) (f* *vt* betar)))
```

...calculate-conductances-and-currents

```
(cond ((> vaf 20000.0)
       (setq gicvbc (f- (f* (f* idumc (f// vabc *vt*))
                            1/alphar))))
      (t
       (setq gicvbc (f* isat (f+ 1.0d0 (f// 1.0d0 betar))
                        (f- (f// (f+ (f- exp-vbe 1.0d0)
                                     (f* exp-vbc (f+ 1.0d0 (f// vbco *vt*)))
                                     -1.0d0)
                                 vaf)
                            (f// exp-vbc *vt*))))))
(setq ico (f* isat vabc
              (f- (f- exp-vbe 1.0d0)
                  (f* 1/alphar
                      (f- exp-vbc 1.0d0))))
      ibo  (f+ (f// (f- idume isat) betaf)
               (f// (f- idumc isat) betar)))))
```

check-convergence

```
(defun-method check-convergence bjt-device-mixin (ichat ibhat)
  (let (tolerance)
      (cond ((not (= 1 *iteration-number*))
             (setq tolerance (f+ *abstol* (f* *reltol* (max (abs ichat)
                                                           (abs ico)))))
             (cond ((<(abs (f- ichat ico)) tolerance)
                    (setq tolerance (f+ *abstol*
                                        (f* *reltol* (max (abs ibhat)
                                                          (abs ibo)))))
                    (cond ((> (abs (f- ibhat ibo)) tolerance)
                           (setq *number-not-converged* (1+ *number-not-converged*)))))
                   (t (setq *number-not-converged* (1+ *number-not-converged*))))))))
```

load-currents-and-conductances

```
(defun-method load-currents-and-conductances bjt-device-mixin ()
  (let ((iceq (f* type (f- ico (f* gicvbe vbeo) (f* gicvbc vbco))))
        (ibeq (f* type (f- ibo (f* gibvbe vbeo) (f* gibvbc vbco)))))
      (sub-from-rhs nc iceq)
      (sub-from-rhs nb ibeq)
      (add-to-rhs   ne iceq ibeq)
      (sub-from-matrix-entry ncnc gicvbc)
      (add-to-matrix-entry   ncnb gicvbe gicvbc)
      (sub-from-matrix-entry ncne gicvbe)
      (sub-from-matrix-entry nbnc gibvbc)
      (add-to-matrix-entry   nbnb gibvbe gibvbc)
      (sub-from-matrix-entry nbne gibvbe)
      (add-to-matrix-entry   nenc gicvbc gibvbc)
      (sub-from-matrix-entry nenb gicvbe gicvbc gibvbe gibvbc)
      (add-to-matrix-entry   nene gicvbe gibvbe)))
```

bjt-device-mixin

```
(defmethod (bjt-device-mixin :calculate-and-load-conductances) ()
```

...bjt-device-mixin

```
  (let ((isat (f* area is))
        vbe vbc delvbc delvbe ichat ibhat)
       (cond ((and (= 1 *iteration-number*) *use-initial-guess*)
              (setq vbeo vcrit
                    vbco 0.0d0))
             (t (setq vbe (f* type (f- (aref *old-sol* nb) (aref *old-sol* ne)))
                      vbc (f* type (f- (aref *old-sol* nb) (aref *old-sol* nc)))
                      delvbe (f- vbe vbeo)
                      delvbc (f- vbc vbco)
                      ichat (f- ico (f* gicvbe delvbe) (f* gicvbc delvbc))
                      ibhat (f- ibo (f* gibvbe delvbe) (f* gibvbc delvbc))
                      vbeo (pnjlim vbe vbeo vcrit)
                      vbco (pnjlim vbc vbco vcrit))))
  (calculate-conductances-and-currents isat)
  (check-convergence ichat ibhat)
  (load-currents-and-conductances)))
```

;bipolar EM1 model flavors

bjt-model-mixin

```
(defflavor bjt-model-mixin ((*type 'npn)
                            (*vaf 1e15)
                            (*is 1e-16)
                            (*bf 100)
                            (*br 1)
                            (*vcrit nil)) (model-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

bjt-model-mixin

```
(defmethod (bjt-model-mixin :load-mna-matrix) ()
  (let* ((is *is)
         (vaf *vaf)
         (betar *br)
         (betaf *bf)
         (vcrit *vcrit)
         (type *type))
        (loop for device in device-list
              do (princ 'curry) (send device ':calculate-and-load-conductances))))
```

bjt-model-mixin

```
(defmethod (bjt-model-mixin :calculate-model-constants) ()
  (cond ((equal *type 'pnp) (setq *type -1.0d0))
        ((equal *type 'npn) (setq *type 1.0d0)))
  (setq *vcrit (f* *vt* (log (f// *vt* 1.4142 *is)))))
```

MOS FLAVORS AND METHODS

mos-device-sparse-matrix-mixin

```
(defflavor mos-device-sparse-matrix-mixin
        (ndnd ndng ndns ndnb ngnd ngng ngns ngnb nsnd nsng nsns nsnb nbnd nbng nbns nbnb) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

mos-capacitances-mixin

```
(defflavor mos-capacitances-mixin
        (vgs-n vgd-n vgb-n cgs-n cgd-n cgb-n cgs-n+1 cgd-n+1 cgb-n+1) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

mos-device-mixin

```
(defflavor mos-device-mixin
        (name nd ng ns nb model-name (lc 1d-6) (wc 1d-6) (ad 1.0) (as 1.0) (pd 0.0) (ps 0.0)
                (vgsi 0.0) (vdsi 0.0) (vbsi 0.0) vgso vdso vbso vbdo vgdo vgbo ido
                ibso ibdo gmo gdso gmbso gbdo gbso device-mode vton vds-sat idrain)
        (mos-device-sparse-matrix-mixin update-self-mixin state-info-mixin mos-capacitances-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

mos-device-mixin

```
(defmethod (mos-device-mixin :set-matrix-pointers) ()
  (setq ndnd (fillij nd nd nil))
  (setq ndng (fillij nd ng nil))
  (setq ndns (fillij nd ns nil))
  (setq ndnb (fillij nd nb nil))
  (setq ngnd (fillij ng nd nil))
  (setq ngng (fillij ng ng nil))
  (setq ngns (fillij ng ns nil))
  (setq ngnb (fillij ng nb nil))
  (setq nsnd (fillij ns nd nil))
  (setq nsng (fillij ns ng nil))
  (setq nsns (fillij ns ns nil))
  (setq nsnb (fillij ns nb nil))
  (setq nbnd (fillij nb nd nil))
  (setq nbng (fillij nb ng nil))
  (setq nbns (fillij nb ns nil))
  (setq nbnb (fillij nb nb nil)))
```

mos-device-mixin

```
(defmethod (mos-device-mixin :update-capacitance-data) ()
```

...mos-device-mixin

```
;;updates the parameters used for meyer's model. at present it is not required because no prediction
;;is being done for the solution at the next time-point. It is being used for the sake of generality
(setq vgs-n vgso
      vgd-n vgdo
      vgb-n vgbo
      cgs-n cgs-n+1
      cgd-n cgd-n+1
      cgb-n cgb-n+1))


(declare (special vto kp gamma lambda js phi type vbi vfb vto tox cgbo cgdo cgso cbs cbd cj cjsw mj mjsw
                  fc*pb f1 f2 f3 f1-sw f2-sw f3-sw))
```

limit-voltages

```
(defun-method limit-voltages mos-device-mixin (vds vgs vgd vbs vbd vcrit idsat issat)
  (cond ((< vdso 0.0d0) (setq vgdo (fetlim vgd vgdo vton)
                              vds  (- vgs vgdo)
                              vdso (- (limvds (- vds) (- vdso)))  ;;new vds assigned to vdso
                              vgso (+ vgdo vdso)))
        (t (setq vgso (fetlim vgs vgso vton)
                 vds  (- vgso vgd)
                 vdso (limvds vds vdso)
                 vgdo  (- vgso vdso))))
  (cond ((< vdso 0.0d0) (setq vcrit (* *vt* (log (// *vt* (* 1.4142 idsat))))
                              vbdo  (pnjlim vbd vbdo vcrit)
                              vbso  (+ vbdo vdso)))
        (t (setq vcrit (* *vt* (log (// *vt* (* 1.4142 issat))))
                 vbso  (pnjlim vbs vbso vcrit)
                 vbdo  (- vbso vdso)))))
```

calculate-source-drain-diode-contributions

```
(defun-method calculate-source-drain-diode-contributions mos-device-mixin (issat idsat)
  (let (evbs evbd)
    (cond ((> vbso 0.0d0) (setq evbs (exp (// vbso *vt*))
                                gbso  (+ *gmin* (// (* issat evbs) *vt*))
                                ibso  (* issat (- evbs 1.0d0))))
          (t (setq gbso (// issat *vt*)
                   ibso (* gbso vbso)
                   gbso (+ gbso *gmin*))))
    (cond ((> vbdo 0.0d0) (setq evbd (exp (// vbdo *vt*))
                                gbdo (+ *gmin* (// (* idsat evbd) *vt*))
                                ibdo (* idsat (- evbd 1.0d0))))
          (t (setq gbdo (// idsat *vt*)
                   ibdo (* gbdo vbdo)
                   gbdo (+ gbdo *gmin*))))))
```

calculate-conductances-and-currents

```
(defun-method calculate-conductances-and-currents mos-device-mixin (vds vgs vbs beta)
```

...calculate-conductances-and-currents

```
(let (vgst betap temp1 temp2)
  (cond ((> vbs 0.0d0) (setq temp1 (sqrt phi)
                             temp1 (max 0.0d0 (- temp1 (// vbs (* 2.0d0 temp1))))))
        (t (setq temp1 (sqrt (- phi vbs)))))
  (setq vton (+ vbi (* gamma temp1))
        vgst (- vgs vton))
  (cond ((> temp1 0.0d0) (setq temp1 (// gamma (* 2.0d0 temp1))))
        (t (setq temp1 0.0d0)))
  (setq vds-sat (max 0.0d0 vgst))
  ;;check region of device operation : saturation or linear
  (setq betap (* beta (+ 1.0d0 (* lambda vds))))
  (cond ((> vgst 0.0d0)                ;device is on
         (cond ((> vgst vds)                ;linear region
                (setq temp2  (* vds (- vgst (* .5d0 vds)))
                      idrain (* betap temp2)
                      gmo    (* betap vds)
                      gdso   (+ (* betap (- vgst vds)) (* lambda beta temp2))
                      gmbso  (* gmo temp1)))
               (t (setq temp2  (* vgst vgst .5d0);saturation region
                        idrain (* betap temp2)
                        gmo    (* betap vgst)
                        gdso   (* lambda beta temp2)
                        gmbso  (* gmo temp1)))))
        (t (setq idrain 0.0d0                ;cut-off region
                 gmo    0.0d0
                 gdso   0.0d0
                 gmbso  0.0d0)))
  (setq ido (- (* device-mode idrain) ibdo))))
```

check-convergence

```
(defun-method check-convergence mos-device-mixin (idhat ibhat)
  (let (tolerance)
    (cond ((not (= 1 *iteration-number*))
           (setq tolerance (+ *abstol* (* *reltol* (max (abs idhat) (abs ido)))))
           (cond ((< (abs (- idhat ido)) tolerance)
                  (setq tolerance (+ *abstol* (* *reltol* (max (abs ibhat) (abs (+ ibso ibdo))))))
                  (cond ((> (abs (- ibhat ibso ibdo)) tolerance)
                         (setq *number-not-converged* (1+ *number-not-converged*)))))
                 (t (setq *number-not-converged* (1+ *number-not-converged*))))))))
```

calculate-capacitance-and-charge

```
(defun-method calculate-capacitance-and-charge mos-device-mixin ()
  (when (and *transient-analysis* (not *use-initial-guess*))
    (let ((cbd-bottom 0.0d0) (cbs-bottom 0.0d0) (cbd-sidewall 0.0d0) (cbs-sidewall 0.0d0)
          capacitance-bd capacitance-bs capacitance-list geq)
      ;;calculate depletion region capacitances
      ;;if cbs and cbd are not given then they are calculated using the junction capacitance
      (cond ((and cbs cbd)
             (setq cbs-bottom cbs
                   cbd-bottom cbd))
```

...calculate-capacitance-and-charge

```
        (t
         (cond (cj
                (setq cbs-bottom (* cj as)
                      cbd-bottom (* cj ad))))))
  (cond (cjsw
         (setq cbs-sidewall (* cjsw ps)
               cbd-sidewall (* cjsw pd))))
  ;;calculate the charge stored in the depletion regions
  (setq capacitance-list (source-drain-junction-charge-and-capacitance q-present-state q-previous-state
                                                                       cbs-bottom cbd-bottom
                                                                  cbs-sidewall cbd-sidewall vbso vbdo)
        ;;this function stores the charge in the state tables and returns a list of capacitance (cbs cbd)
        capacitance-bs (first capacitance-list)
        capacitance-bd (second capacitance-list)
        geq (first (get-capacitor-contribution 'cbs capacitance-bs q-previous-state q-present-state
                                               i-previous-state i-present-state))
        gbso (+ gbso geq)
        ibso (+ ibso (get-state-info i-present-state 'cbs))
        geq (first (get-capacitor-contribution 'cbd capacitance-bd q-previous-state q-present-state
                                               i-previous-state i-present-state))
        gbdo (+ gbdo geq)
        ido  (+ ido ibdo)
        ibdo (+ ibdo (get-state-info i-present-state 'cbd))
        ido  (- ido ibdo)))

;;calculate meyer-capacitances for a mos device
(let ((cox (// (* wc lc 3.453d-11) tox))
      (cgb-overlap (* cgbo lc))
      (cgs-overlap (* cgso wc))
      (cgd-overlap (* cgdo wc))
      capacitance-list)
  (cond ((equal 1 device-mode)          ;normal mode of operation
    (setq capacitance-list (calculate-meyer-capacitances vgso vdso vbso vds-sat vton cox cgb-overlap
                                                         cgs-overlap cgd-overlap)
          cgs-n+1 (first capacitance-list)
          cgd-n+1 (second capacitance-list)
          cgb-n+1 (third capacitance-list)))
        (t                              ;inverse mode of operation (interchange drain and source)
    (setq capacitance-list (calculate-meyer-capacitances vgdo (- vdso) vbdo vds-sat vton cox cgb-overlap
                                                         cgd-overlap cgs-overlap)
          cgd-n+1 (first capacitance-list)
          cgs-n+1 (second capacitance-list)
          cgb-n+1 (third capacitance-list))))
  ;;calculate charges
  (when (and (= 1 *number-timepoints*) (= 1 *iteration-number*))
    (setq cgs-n cgs-n+1
          cgd-n cgd-n+1
          cgb-n cgb-n+1
          vgs-n vgso
          vgd-n vgdo
          vgb-n vgbo))
```

...calculate-capacitance-and-charge

```
(calculate-and-store-charges-for-meyer-model cgs-n cgd-n cgb-n cgs-n+1 cgd-n+1 cgb-n+1
                                             vgs-n vgd-n vgb-n vgso vgdo vgbo
                                             q-previous-state q-present-state))))
```

load-currents-and-conductances

```
(defun-method load-currents-and-conductances mos-device-mixin (normal inverse)
  (let ((gcgs 0.d0) (gcgd 0.0d0) (gcgb 0.0d0)
        (ieqcgs 0.0d0) (ieqcgd 0.0d0) (ieqcgb 0.0d0)
        ieqbs ieqbd idreq)
    (cond ((and *transient-analysis* (not *use-initial-guess*))
           ;;integrate to get equivalent conductance and current
           (setq gcgs (first (get-capacitor-contribution 'cgs cgs-n+1 q-previous-state q-present-state
                                                         i-previous-state i-present-state))
                 gcgd (first (get-capacitor-contribution 'cgd cgd-n+1 q-previous-state q-present-state
                                                         i-previous-state i-present-state))
                 gcgb (first (get-capacitor-contribution 'cgb cgb-n+1 q-previous-state q-present-state
                                                         i-previous-state i-present-state))
                 ieqcgs (* type (- (get-state-info i-present-state 'cgs) (* gcgs vgso)))
                 ieqcgd (* type (- (get-state-info i-present-state 'cgd) (* gcgd vgdo)))
                 ieqcgb (* type (- (get-state-info i-present-state 'cgb) (* gcgb vgbo))))))
    (setq ieqbs (* type (- ibso (* (- gbso *gmin*) vbso)))
          ieqbd (* type (- ibdo (* (- gbdo *gmin*) vbdo))))
    (cond ((= 1 device-mode)
           (setq idreq (* type (- idrain (* gdso vdso) (* gmo vgso) (* gmbso vbso)))))
          (t (setq idreq (- (* type (- idrain (* gdso (- vdso)) (* gmo vgdo) (* gmbso vbdo)))))))
    ;;load rhs and mna matrix
    (sub-from-rhs ng ieqcgs ieqcgd ieqcgb)
    (sub-from-rhs nb ieqbs ieqbd (- ieqcgb))
    (add-to-rhs nd  ieqbd ieqcgd (- idreq))
    (add-to-rhs ns  idreq ieqbs ieqcgs)
    (add-to-matrix-entry  ngng gcgs gcgd gcgb)
    (add-to-matrix-entry  nbnb gbdo gbso gcgb)
    (add-to-matrix-entry  ndnd gdso gbdo (* inverse (+ gmo gmbso)) gcgd)
    (add-to-matrix-entry  nsns gdso gbso (* normal (+ gmo gmbso)) gcgs)
    (sub-from-matrix-entry ngnd gcgd)
    (sub-from-matrix-entry ngns gcgs)
    (sub-from-matrix-entry ngnb gcgb)
    (sub-from-matrix-entry nbnd gbdo)
    (sub-from-matrix-entry nbng gcgb)
    (sub-from-matrix-entry nbns gbso)
    (sub-from-matrix-entry ndng (* (- inverse normal) gmo) gcgd)
    (sub-from-matrix-entry ndnb gbdo (* (- inverse normal) gmbso))
    (sub-from-matrix-entry ndns gdso (* normal (+ gmo gmbso)))
    (sub-from-matrix-entry nsng (* (- normal inverse) gmo) gcgs)
    (sub-from-matrix-entry nsnb gbso (* (- normal inverse) gmbso))
    (sub-from-matrix-entry nsnd gdso (* inverse (+ gmo gmbso)))))
```

mos-device-mixin

```
(defmethod (mos-device-mixin :calculate-and-load-conductances) ()
```

...mos-device-mixin

```
(let ((idsat (* ad js))
      (issat (* as js))
      (beta  (// (* kp wc) lc))
      vds vgs vbs vcrit vbd vgd delvbs delvbd delvgs delvds delvgd ibhat idhat inverse normal)
  (cond ((and (= 1 *iteration-number*) *use-initial-guess*)
         (setq vdso 0.0d0
               vgso vto
               vbso -1.0d0))
        (t (setq vds (* type (- (aref *old-sol* nd) (aref *old-sol* ns)))
                 vgs (* type (- (aref *old-sol* ng) (aref *old-sol* ns)))
                 vbs (* type (- (aref *old-sol* nb) (aref *old-sol* ns)))
                 vbd (- vbs vds)
                 vgd (- vgs vds)
                 delvbs (- vbs vbso)
                 delvbd (- vbd vbdo)
                 delvgs (- vgs vgso)
                 delvds (- vds vdso)
                 delvgd (- vgd vgdo)
                 ibhat  (+ ibso ibdo (* gbdo delvbd) (* gbso delvbs)))
           (cond ((= 1 device-mode) (setq idhat (+ ido (- (* gbdo delvbd)) (* gmbso delvbs)
                                                  (* gmo delvgs) (* gdso delvds))))
                 (t (setq idhat (- ido (* (- gbdo gmbso) delvbd) (* gmo delvgd)
                                   (- (* gdso delvds))))))
           ;;limit nonlinear branch voltages
           (limit-voltages vds vgs vgd vbs vbd vcrit idsat issat)))
  (setq vbdo (- vbso vdso)
        vgdo (- vgso vdso)
        vgbo (- vgso vbso))
  (calculate-source-drain-diode-contributions issat idsat)
  ;; check for mode of device operation: normal or inverse and calculate conductances
  (cond ((< vdso 0.0d0)                ;inverse region of operation
         (setq device-mode -1
               inverse 1
               normal  0)
         (calculate-conductances-and-currents (- vdso) vgdo vbdo beta))
        (t (setq device-mode 1
                 inverse 0
                 normal  1)
           (calculate-conductances-and-currents vdso vgso vbso beta)))
  (calculate-capacitance-and-charge)
  ;;check convergence
  (check-convergence idhat ibhat)
  ;;load current vector and conductances
  (load-currents-and-conductances normal inverse)))
```

;;MOS1 MODEL FLAVORS . WILL REQUIRE DIFFERENT FLAVORS FOR A DIFFERENT MOS MODEL

mos1-model-mixin

...mos1-model-mixin

```
(defflavor mos1-model-mixin ((*type 'nmos)
                             (*vto 1.0d0)
                             (*kp 2.0d-5)
                             (*gamma 0.0d0)
                             (*lambda .0d0)
                             (*js 1d-14)
                             (*phi .6d0)
                             (*tox 1d-7)
                             (*cgso 0.0d0)
                             (*cgdo 0.0d0)
                             (*cgbo 0.0d0)
                             (*cbs nil)
                             (*cbd nil)
                             (*cj nil)
                             (*cjsw nil)
                             (*mj .5d0)
                             (*mjsw (// 1.0d0 3.0d0))
                             (*pb 0.8d0)
                             (*fc 0.5d0)
                             (*vfb nil)
                             (*vbi nil)
                             (*fc*pb 0.4d0)
                             (*f1 nil) (*f2 nil) (*f3 nil)
                             (*f1-sw nil) (*f2-sw nil) (*f3-sw nil))(model-mixin)
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

mos1-model-mixin

```
(defmethod (mos1-model-mixin :load-mna-matrix-and-rhs) ()
  (let* ((kp *kp)
         (gamma *gamma)
         (lambda *lambda)
         (js *js)
         (phi *phi)
         (tox *tox)
         (type *type)
         (vfb *vfb)
         (cgbo *cgbo)
         (cgdo *cgdo)
         (cgso *cgso)
         (cbs *cbs)
         (cbd *cbd)
         (cj *cj)
         (cjsw *cjsw)
         (mj *mj)
         (mjsw *mjsw)
         (pb *pb)
         (fc*pb *fc*pb)
         (f1 *f1)
         (f2 *f2)
```

...mos1-model-mixin

```
        (f3 *f3)
        (f1-sw *f1-sw)
        (f2-sw *f2-sw)
        (f3-sw *f3-sw)
        (vbi (* type *vbi))
        (vto (* type *vto)))
    (loop for device in device-list
          do (send device ':calculate-and-load-conductances))))
```

mos1-model-mixin

```
(defmethod (mos1-model-mixin :calculate-model-constants) ()      ;some constants to be used during analysis
  ;;calculate the magnitude of vfb for charge storage calculations
  (cond ((equal *type 'pmos) (setq *type -1.0
                                   *vbi (+ *vto (* *gamma (sqrt *phi)))
                                   *vfb (- (- *vto) *phi (* *gamma (sqrt *phi)))))
        ((equal *type 'nmos) (setq *type 1.0
                                   *vbi (- *vto (* *gamma (sqrt *phi)))
                                   *vfb (- *vto *phi (* *gamma (sqrt *phi))))))
  ;;calculate the constant coefficients for forward bias depletion capacitances
  ;;the constants are defined as
  ;;        f1 = pb * (1 - (1 - fc)^ (1 - mj))/(1 - mj)
  ;;        f2 = 1 / (1 - fc)^ mj
  ;;        f3 = mj / (pb * (1 - fc)^ (1 + mj)) = mj * f2 / (pb * (1 - fc))
  ;;f1-sw, f2-sw, and f3-sw are similarly defined except that mjsw is used instead of mj
  (setq *f1 (* *pb (// (^ (- 1.0d0 (- 1.0d0 *fc)) (- 1.0d0 *mj)) (- 1.0d0 *mj)))
        *f2 (^ (- 1.0d0 *fc) (- *mj))
        *f3 (// (* *mj *f2) (* *pb (- 1.0d0 *fc)))
        *f1-sw (* *pb (// (^ (- 1.0d0 (- 1.0d0 *fc)) (- 1.0d0 *mjsw)) (- 1.0d0 *mjsw)))
        *f2-sw (^ (- 1.0d0 *fc) (- *mjsw))
        *f3-sw (// (* *mjsw *f2) (* *pb (- 1.0d0 *fc)))
        *fc*pb (* *fc *pb)))
```

update-mos-capacitance-data

```
(defun update-mos-capacitance-data ()
  (loop for mos in *mos-list*
        do (send mos ':update-capacitance-data)))
```

calculate-and-store-charges-for-meyer-model

```
(defun calculate-and-store-charges-for-meyer-model (cgs-n cgd-n cgb-n cgs-n+1 cgd-n+1 cgb-n+1
                                                    vgs-n vgd-n vgb-n vgs-n+1 vgd-n+1 vgb-n+1
                                                    q-previous-state q-present-state)
  ;;this function calculates the charge of a nonlinear capacitor using an approximate integration
  ;;of C(v) wrt v. Trapezoidal integration is used whereby
  ;;          q-n+1 = q-n + (.5 * (c-n + c-n+1) (v-n+1 - v-n))      for n  0
  ;;          q-0 = 0
  (let (qcgs qcgd qcgb)
    (cond ((and (= 1 *number-timepoints*) (= 1 *iteration-number*))
      (setq qcgs (* cgs-n+1 vgs-n+1)
```

...calculate-and-store-charges-for-meyer-model

```
          qcgd (* cgd-n+1 vgd-n+1)
          qcgb (* cgb-n+1 vgb-n+1))
         (store-state-info q-previous-state 'cgs qcgs)
         (store-state-info q-previous-state 'cgd qcgd)
         (store-state-info q-previous-state 'cgb qcgb))
        (t
         (setq qcgs (+ (* .5 (+ cgs-n+1 cgs-n) (- vgs-n+1 vgs-n)) (get-state-info q-previous-state 'cgs))
               qcgd (+ (* .5 (+ cgd-n+1 cgd-n) (- vgd-n+1 vgd-n)) (get-state-info q-previous-state 'cgd))
               qcgb (+ (* .5 (+ cgb-n+1 cgb-n) (- vgb-n+1 vgb-n)) (get-state-info q-previous-state 'cgb)))))
   (store-state-info q-present-state 'cgs qcgs)
   (store-state-info q-present-state 'cgd qcgd)
   (store-state-info q-present-state 'cgb qcgb)))
```

calculate-meyer-capacitances

```
(defun calculate-meyer-capacitances (vgs vds vbs vds-sat vton cox cgb-overlap cgs-overlap cgd-overlap)
  ;; the regions of operation of the device are
  ;; cut-off       vgs <= vfb + vbs
  ;; sub-threshold  vfb + vbs < vgs <= vton
  ;; on            vton < vgs
  ;;              saturation vds >= vdsat
  ;;              linear     vds < vdsat
  (let ((vgst (- vgs vton))
        (vt-minus-vfb-vbs (- vton vfb vbs))
        cgb cgs cgd vdiff vdiff1 vdiff-square)
    (cond ((<= vgst (- vt-minus-vfb-vbs))                ;i.e. vgb <= vfb the device is cutoff
           (setq cgb (+ cox cgb-overlap)
                 cgs cgs-overlap
                 cgd cgd-overlap))
          ((<= vgst 0.0d0)
           (setq cgb (+ (- (// (* vgst cox) vt-minus-vfb-vbs)) cgb-overlap)
                 cgs cgs-overlap
                 cgd cgd-overlap))
          (t                              ;device is on
           (cond ((<= vds-sat vds)                ;in saturation region
                  (setq cgb cgb-overlap
                        cgs (+ (// cox 1.5) cgs-overlap)
                        cgd cgd-overlap))
                 (t                              ;in linear region
                  (setq vdiff (- (* 2 vds-sat) vds)
                        vdiff-square (* vdiff vdiff)
                        vdiff1 (- vds-sat vds 1.0d-12)
                        cgb cgb-overlap
                        cgs (+ (// (* cox (- 1.0d0 (// (* vdiff1 vdiff1) vdiff-square))) 1.5) cgs-overlap)
                        cgd (+ (// (* cox (- 1.0d0 (// (* vds-sat vds-sat) vdiff-square))) 1.5) cgd-overlap))))))
    (list cgs cgd cgb)))
```

source-drain-junction-charge-and-capacitance

```
(defun source-drain-junction-charge-and-capacitance
  (q-present-state q-previous-state cbs-bottom cbd-bottom cbs-sidewall cbd-sidewall
```

...source-drain-junction-charge-and-capacitance

```
 vbs vbd)
 ;;the drain source depletion capacitances and charges are calculated
 (let (qc-list qbs qbd cbs cbd)
   (setq qc-list (depletion-region-charge-and-capacitance vbs cbs-bottom cbs-sidewall)
         qbs (first qc-list)
         cbs (second qc-list)
         qc-list (depletion-region-charge-and-capacitance vbd cbd-bottom cbd-sidewall)
         qbd (first qc-list)
         cbd (second qc-list))
   (when (and (= 1 *number-timepoints*) (= 1 *iteration-number*))
     (store-state-info q-previous-state 'cbs qbs)
     (store-state-info q-previous-state 'cbd qbd))
   (store-state-info q-present-state 'cbs qbs)
   (store-state-info q-present-state 'cbd qbd)
   (list cbs cbd)))
```

depletion-region-charge-and-capacitance

```
(defun depletion-region-charge-and-capacitance (vc cj0 cj0-sw)
  ;;this function calculates the charge in a depletion region. if cj0-sw, the sidewall capacitance is
  ;;zero then no sidewall capacitance is calculated
  ;;in the case of a forward biased junction 3 precalculated model constants are used
  ;;they are
  ;;      f1 = pb * (1 - (1 - fc)^ (1 - mj))/(1 - mj)
  ;;      f2 = 1 / (1 - fc)^ mj
  ;;      f3 = mj / (pb * (1 - fc)^ (1 + mj)) = mj * f2 / (pb * (1 - fc))
  ;;f1-sw, f2-sw, and f3-sw are similarly defined except that mjsw is used instead of mj
  ;;these constants are calculated in :method :calculte-model-constants
  (let (temp arg arg-sw (qj 0.0d0) (cj 0.0d0))
    (cond ((and (= 0.0d0 cj0) (= 0.0d0 cj0-sw)) (list qj cj))
          (t
           ;;check if the junction is reverse biased or forward biased
           (cond ((< vc fc*pb)                    ;reverse biased
                  (setq temp (- 1.0d0 (// vc pb)))
                  (when (not (= 0.0d0 cj0))       ;if bottom capacitance
                        (setq arg  (^  temp (- mj))
                              qj  (* pb (// (* cj0 (- 1.0d0 (* temp arg))) (- 1.0d0 mj)))
                              cj  (* cj0 arg)))
                  (when (not (= 0.0d0 cj0-sw))    ;add sidewall contribution if cj0-sw  0
                        (setq arg-sw (^  temp (- mjsw))
                              qj (+ qj (* pb (// (* cj0-sw (- 1.0d0 (* temp arg-sw))) (- 1.0d0 mjsw))))
                              cj (+ cj (* cj0-sw arg-sw))))
                  (list qj cj))
                 (t                               ;forward biased
                  (let ((vc-minus-fc*pb (- vc fc*pb)))
                    (when (not (= 0.0d0 cj0))     ;bottom capacitance
                          (setq cj (* cj0 (+ f2 (* vc-minus-fc*pb f3)))
                                qj (* cj0 (+ f1 (* vc-minus-fc*pb (+ f2 (* .5 vc-minus-fc*pb f3)))))))
                    (when (not (= 0.0d0 cj0-sw)) ;add sidewall contributions
                          (setq cj (+ cj (* cj0-sw (+ f2-sw (* vc-minus-fc*pb f3-sw))))
                                qj (+ qj (* cj0-sw (+ f1-sw (* vc-minus-fc*pb (+ f2-sw
```

...depletion-region-charge-and-capacitance

```
                                                    (* .5 vc-minus-fc*pb f3-sw)))))))
(list qj cj)))))))
```

REFERENCES

[1] K. Mayaram and D. O. Pederson, "Circuit Simulation in LISP", *Digest of Technical Papers ICCAD-84*, Santa Clara, California, pp. 24-26, Nov. 1984.

[2] K. Mayaram and D. O. Pederson, "Circuit Simulation in LISP", *Memo. No. UCB/ERL M84/60*, Electronics Research Laboratory, University of California, Berkeley, Aug. 1984.

[3] E. Horowitz, *Fundamentals of Programming Languages*, Computer Science Press, 1984.

[4] D. Robinson, "Object-Oriented Software Systems", *BYTE*, Vol. 6, No. 8, pp. 74-86, Aug. 1981.

[5] D. Weinreb and D. Moon, *LISP Machine Manual*, Symbolics Inc., 1981.

[6] The Xerox Learning Research Group, "The Smalltalk-80 System", *BYTE*, Vol. 6, No. 8, pp. 36-48, Aug. 1981.

[7] A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, Masachussets, 1984.

[8] H. Cannon, "A non-hierarchical approach to object-oriented programming", Unpublished paper, Artificial Intelligence Laboratory, MIT, Cambridge.

[9] E. Cohen, "Program Reference For SPICE2", *Memo. No. ERL-M592*, Electronics Research Laboratory, University of California, Berkeley, June 1976.

[10] T. Vucurevich, *Private Communication.*