Bell Laboratories
Lucent Technologies

subject: Solving Partial Differential Equations with the `PROPHET` simulator

date: December 5, 1997

from: Conor S. Rafferty
MH 2D-303B
908 582 3575
`conor@lucent.com`

R. Kent Smith
MH 2D-303C
908 582 7522
`kentsmith@lucent.com`

Abstract

The PROPHET simulator is a framework to solve systems of partial differential equations (PDEs) in time and 1,2 or 3 space dimensions. The simulator is designed with three main goals: efficiency, geometric flexibility, equation extensibility. The first two distinguish it from canned packages such as Mathematica, which do not allow the use of arbitrary shapes or grids and are not tuned to solve systems with 10^5 or 10^6 unknowns. The third distinguishes it from previous application-specific simulators such as PISCES or SUPREM-4. The simulator has been used in production for several years to predict semiconductor manufacturing processes.

This memorandum focusses on the third goal, the ability to add new equations to the simulator. It describes the language used to represent the equations, the available operators, the available controls for influencing the solution process, and the procedure for adding new operators.

1. Introduction

The PROPHET simulator is a framework to solve systems of partial differential equations (PDEs) in time and 1,2 or 3 space dimensions. (Ordinary differential equations can be solved as a degenerate case). The PDEs are discretized using either finite elements or finite volume methods in space and with implicit methods in time, reducing the differential equations to a large system of algebraic equations. At each timestep the algebraic equations are solved by Newton's method. The matrix resulting from the linearization is solved by sparse iterative or direct methods.

The simulator is intended for use in production applications, and is therefore designed with maximum efficiency as a goal. The matrix assembly code takes advantage of the vector hardware capabilities on machines such as the Cray YMP, allowing the solution of 3D problems in reasonable time. Two dimensional problems are routinely solved on a workstation. At Bell Labs, PROPHET uses the Bell Labs Sparse Matrix Package (BLSMP) matrix solver for maximum efficiency in the solution phase; release versions are interfaced to the SLES matrix solver from Argonne which provides acceptable performance. Other solvers can easily be interfaced.

A second major concern is to solve problems on arbitrary geometries. To this end, the discretization library makes no assumptions about the domain but uses arbitrary combinations of elements to describe its shape, such as triangles and quadrilaterals in 2D, tetrahedra, bricks or prisms in 3D. There are limited built-in grid generation facilities, but arbitrary grids can be read from other sources and calculations performed on those domains.

The final major design goal is to allow new equations to be specified and solved by a user or a model developer who may not be familiar with numerical methods. This memorandum focusses on the third goal, the ability to add new equations and even new types of equations to the simulator. This goal is particularly important for process simulation, where there is no consensus on the underlying equations describing solid-state diffusion, and new models are frequently proposed. As a very simple example of setting up equations, the system

$$\begin{aligned} \nabla^2 Y + 100Y &= 0 \\ Y(0) &= 1 \\ Y(1) &= 1 \end{aligned}$$

can be described with the syntax

```
system name=test4
+   sysvars=y
+   nterm=3
+   term0=-1*box_div.lapflux(y|y)@{silicon}
+   term1=dirichlet.default_dirichlet(0|y)@{silicon/bulk,silicon/exposed}
+   term2=100.0e0*nodal.copy(y|y)@{silicon}
```

Typically there is one line of description for each term in the PDE, and one line for each boundary condition. The overall system is described in terms of a number of building blocks, each of which takes as input either primary variables or temporary variables which can be created on the fly as functions of other variables. Standard building blocks include the Laplacian, the drift operator, the nodal lumping operator, dirichlet and radiative boundary conditions, the time derivative, and so on.

PROPHET is designed so that new building blocks can be easily constructed.

The intention is that the code can be used by three categories of users: application users, model developers, and primary developers. Application users are primarily interested in running the code off-the-shelf but are interested in having some control over its execution, such as the choice of linear method. Model developers are interested in adding new equations, either by combining existing operators or building new operators according to specification. The primary developers are concerned with the underlying database, the implementation of the discretization library and the linear solvers, the grid structure, and so on. A block diagram of the simulator is shown below.

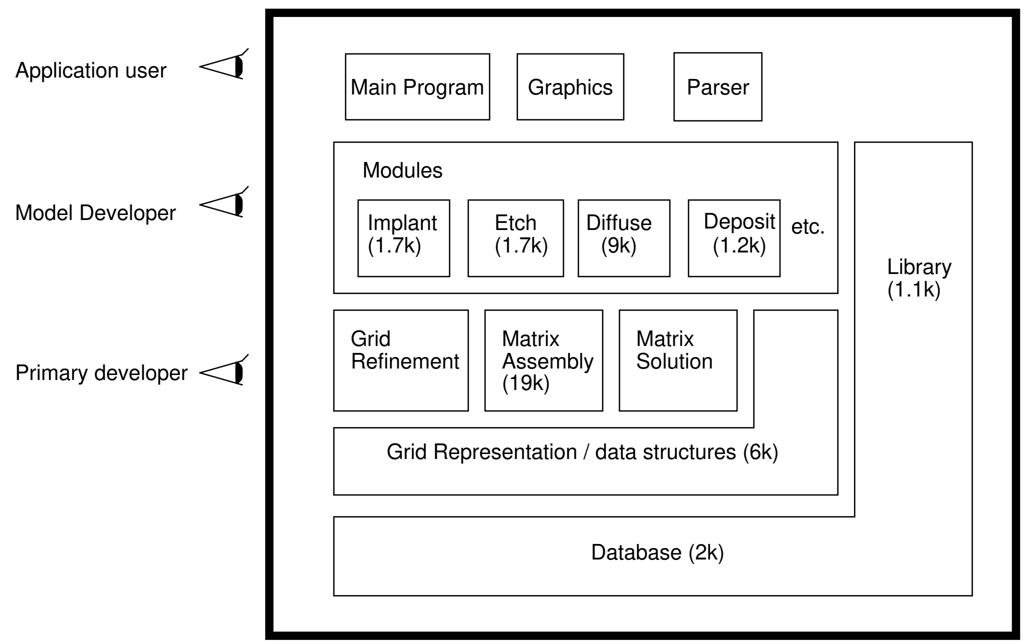

Figure 1. Overall structure of PROPHET. The simulator is built on two major blocks, a hierarchical database and a PDE solving library. Extra modules to configure it as a process simulator are layered on top. Numbers in parenthesis indicate lines of code.

In the subsequent sections, the decomposition of a PDE into its component parts is described, along with its expression in terms of existing operators in the simulator. The existing operators are listed and described. In the cases where new operators must be defined, the subroutine interface to such operators is described.

2. Terminology and PDE Decomposition

The basic problem unit in PROPHET is the `pdeblock` or block of coupled partial differential equations. The equations are to be solved on a `domain` which contains a number of `fields`. Fields may either be fixed fields, if there are no equations to be solved for them, or variable fields, abbreviated as variables.

A block of equations is square; it necessarily has the same number of variables and equations. A block can then be considered as an assembly of sub-terms, called `pdeterms`; for instance a reaction-diffusion system might consist of several reaction terms between various species, a diffusion term for each species, and a transient term for each species. A term-by-term decomposition of point defect diffusion equations in silicon is given schematically in Figure 2. The form of Figure 2 defines the sign convention in PROPHET; all terms in an equation are added to create a residual which will be converged to zero by a Newton iteration.

$$\begin{aligned} \frac{\partial C_I}{\partial t} + \boxed{\nabla \bullet (D_I \nabla C_I)} + k(C_I C_V - C_I^* C_V^*) &= 0 \\ \frac{\partial C_V}{\partial t} + \boxed{\nabla \bullet (D_V \nabla C_V)} + k(C_I C_V - C_I^* C_V^*) &= 0 \end{aligned}$$

Figure 2. A block of equations representing point defect diffusion in silicon (without impurities). The block is decomposed into several terms: two laplacian terms, one binary-recombination term, and a transient term.

The terms can be rectangular, not necessarily square, since they define a number *n* of outputs in terms of *m* inputs. Each of these `pdeterms` can be further subdivided into a geometrical part and a physical part (a `geoterm` and a `phyterm`). In the above instance, the laplacian operator can be considered as the combination of the divergence operator $\nabla \bullet$ and the flux of point defects $D \nabla C$. The discretization defines the geometrical operators and the means of evaluating the gradients in the flux; these would be handled differently in a finite element or finite volume implementation. The construction of the flux from the gradients is, however, independent of the discretization and is defined in terms of textbook constructs such as the concentration of a quantity at a point in space and the gradient of a quantity in space, a scalar and a vector respectively. The *numerical* work of discretizing the equations, and the *modeling* work of defining fluxes in terms of concentrations and gradients are thus separated in PROPHET; the separation can be symbolically written as follows.

$$\text{PDEBLOCK} = \underbrace{G_1 P_1}_{\text{PDETERM1}} + \underbrace{G_2 P_2}_{\text{PDETERM2}} + \underbrace{G_3 P_3}_{\text{PDETERM3}}$$

3. System definition using the system command

The "`system`" command defines a new set of equations to be solved. Each system must have a name, which is later referenced by the "`solve`" command.

The system gives two lists of variables. The first is the system variables, "`sysvars`", for which differential equations are to be solved. The second list is called "`tmpvars`" and lists any temporary variables used in formulating the full system.

The terms of the system are then listed. The number of terms is given first ("`nterm`") followed by each term (“`term0,term1,...`”), starting at zero. The format of the terms is as follows.

optional multiplier factor

R*`geoterm.phyterm( I1,I2,I3|O1,O2,O3)@{region1,region2}`

The geometric operator of the term

The physical operator of the term

Input Fields

Output Fields

Materials or interfaces to which term applies

The outputs of the term must be the names of system variables. The input variables can be system variables, temporary variables, or other fields which exist in the structure but which are not currently being solved.

The same syntax is used to specify both bulk operators and boundary conditions, as exhibited in the specification of the dirichlet boundary conditions in the example of the introduction.

If temporary variables are defined, then a similar list of functions is given, with “`nfunction`” and “`func0, func1,...`” being the list of functions. The functions are defined with the same syntax, without a `geoterm`. The output variables of a function must be temporary variables, while the input variables can be system variables, temporary variables, or other fields which exist in the structure which are not being solved. For example,

$$\frac{\partial N}{\partial t} = \nabla^2 \sqrt{N}$$

could be written

```
system name=ttt
+ sysvars=n
+ tmpvars=sn
+ nterm=2
+ term0=box_div.lapflux(sn|n)@{silicon}
+ term1=transient.ddt(n|n)@{silicon}
+ nfunc=1
+ func0=sqroot(n|sn)@{silicon}
```

The optional multiplier factor `R` multiplies the entire term. This is best used for setting the sign of a term rather than providing coefficients to a PDE, since there is no way to specify that the multiplier is different in one region or another.

When functions are given, usually each function generates one variable, though it is also possible for a function to simultaneously generate several variables. If a function takes as input other variables which are in turn functions of system variables, the order of function specification is important. The “first-level” functions which are defined in terms of system variables should be defined first, then the “second-level” functions, which are functions of only system variables and “first-level” variables, should be given, and so on.

4. System definition via the PROPHET database

The system command is all that is necessary for building basic systems. This section describes a little more of the internal workins in order to access more advanced system features such as permanent systems, template systems and modifying numerical parameters. It can be skipped on first reading.

Systems which are used frequently can be stored in the PROPHET database for access by other users. In fact, the solver always uses a database description of a system; the system command does its work by modifying the in-memory copy of the database.

The database has a tree starting at library/math/systems. For each named system, for instance ABC, there is an entry library/math/systems/LABEL/math.ABC/pde. Normally the label is the same as the system name. If no system is named on the solve command, but there is a system .../math.default/pde, then that default system will be used in solving its variables.

For the system ttt described previously, the database entries would appear as follows. It contains the same information in a more verbose format.

```
[library/math/systems/ttt/math.ttt/] pde = (list) {
        SeeAlso = (string) "/library/math/systems/default_numerical_parameters";
        order = (string) "n";
        spawn = (string) "n,sn";
        nterm = (int) 2;
        term0 = (list)
        {
                geoterm = (string) "box_div";
                phyterm = (string) "lapflux";
                equations = (string) "n";
                variables = (string) "sn";
                sign = (real) 1;
        };
        term1 = (list) {
                geoterm = (string) "transient";
                phyterm = (string) "ddt";
                equations = (string) "n";
                variables = (string) "n";
                sign = (real) 1;
                };
        elimination = (list) {
                order = (string) "sn";
                nterm = (int) 1;
                term0 = (list) {
                        geoterm = (string) "dummy";
                        phyterm = (string) "sqrt";
                        equations = (string) "sn";
                        variables = (string) "n";
                        sign = (real) 1;
                };
        };
};
```

The property pde is a list with the properties spawn, order, nterm and termNNN and optionally elimination.

The property order is a list of the system variables and corresponds to the sysvars keyword on the system command. As its name implies, the order is significant. The order in which the names appear dictates the order in which the variables which will appear in the matrix, which influences the block fill structure of the matrix and may affect convergence if the matrix is solved iteratively. Generally variables which are linked to many other variables should appear late in the list, so that variables with few links can be eliminated early.

The property "`spawn`" is a list of field names which should be created on the domain when this sytem is solved, if they are not already present. The `system` command by default includes all the system variables and temporary variables on the `spawn` list, but it can be overriden by giving an explicit `spawn` list, or by using a `dbase` command after the `system` command. There are cases where the default behavior is not desired, related to setting up general systems which refer to many variables, only a few of which are present in the current domain. For instance, if a system solves for "`boron,phosphorus,arsenic`" but only "`boron`" is present in the domain, it would not be desirable to generate `phosphorus` and `arsenic`. In that case the `spawn` list would be empty. Another feature of the "`spawn`" variable is conditional creation. A variable may be marked as being necessary if another variable exists. An example of this usage is when distinguishing between active and chemical species. A variable for the active species should only be generated if the chemical species is present. The syntax for that is

```
spawn="psi,boron*:boron,arsenic*:arsenic,phosphorus*:phosphorus,antimony*:antimony"
```

Thus `psi` is always generated, but the active concentrations `boron*,arsenic*`, etc are created only as necessary.

The relation between the variable lists on the `system` command and the variable lists in the PDE property list is tabulated below.

`System` command	`pde` property list
`sysvars`	`order`
`tmpvars`	`elimination/order`
`[spawn=sysvars+tmpvars]`	`spawn`

The property `nterm` is an integer specifying the number of terms. The properties `termNNN` are each lists, which should contain the properties `geoterm`, `phyterm`, `equations`, `variables` and `sign`. The `geoterm` and `phyterm` properties are the names of the appropriate subroutines which implement this term. The `equations` property is a string containing the names of the outputs of this term and the `variables` property is a string containing the inputs of the term. The `sign` is a real multiplier for the entire term; in spite of its name it can contain any real numberl. For boundary terms, there are additional properties, all called `between,` which have a string value of the form "`material1/material2`". The boundary term is applied to the interfaces between `material1` and `material2` (where `material2` can also be the name of a surface, like `exposed` or `anode`).

The `elimination` property on the `pde` list is another list similar in structure to the `pde` list itself. It has an `order` variable which defines the temporary variables to be created, corresponding to the `tmpvars` keyword of the `system` command. Then there is an `nterm` corresponding to the `nfunc` of the `system` command, and a series of terms defining the functions. The only novelty here is that these functions have no geometry associated with them; the `geoterm` is always dummy.

Finally, a pde list should have the following numerical properties. They are usually inherited by a SeeAlso link, as in the above example.

```
default_numerical_parameters = (list) {
maxNewton = (int) 10;           #max number of newton loops
NewtonUpd = (real) 1e-5;        #newton convergence criterion

time.min = (real) 1e-08;        #minimum allowed timestep
time.extend = (real) 1.1;       #maximum timestep stretch
time.init = (real) 1e-06;       #starting timestep

#expert parameters
topNewton = (real) 0.01;        #loosest linear tolerance
botNewton = (real) 1e-07;       #tighest linear tolerance
NewtonChk = (real) 0;           #when to consider early exit
NewtonRhs = (real) 0;           #when to take early exit
NewtonMaxUpd = (real) 1e+09;    #largest reasonable update
};
```

To change the maximum number of Newton loops, the user could issue the command

```
dbase modify name=library/math/systems/ttt/math.ttt/pde/maxNewton ival=100
```

or even

```
dbase modify name=library/math/systems/default_numerical_parameters/maxNewton ival=100
```

which would change the parameter for all systems, not just ttt.

At present the pde definition itself does not specify to what regions it should be applied. That information is carried in the existence of a SeeAlso link
from library/physics/material/field/SeeAlso
to library/math/systems/ABC

The solver queries the existence of library/physics/material/field/math.ABC and if it finds it by following the link, the system is solved in that region. If not, the field is left alone in that region. If no field has a system defined, it is not an error, but no work is done. This usually happens when a system is defined but on the solve command the name of the system is omitted or mis-spelled.

5. Existing modules

Having described how to assemble operators, the available operators are now specified.

5.1 Geometrical operators ("geoterms").

`fel_div`	Finite element discretization of the divergence operator
`box_div`	Finite volume discretization of the divergence operator
`volume (or nodal)`	Finite volume discretization of the nodal-weighting operator
`interface`	Finite volume discretization for interface fluxes
`dirichlet`	Enforce dirichlet boundary conditions
`transient`	Nodal operator which can access historical values on the domain
`constraint`	Specifies constraints between variables at an interface

5.1.1 Divergence operator

The divergence operator is used in constructing such terms as $\nabla\bullet F$, where the flux F is defined as a function of various solution variables a, b, c and their gradients.

$$F = F(a, b, c, \nabla a, \nabla b, \nabla c)$$

The finite volume ("box") discretization is much faster and is recommended. The current implementation, both box and finite element, codes the negative of the divergence. This can be adjusted with the sign parameter as desired. The negative sign is usually desired in order to obtained the expected behavior in the diffusion equation.

$$\frac{\partial C}{\partial t} - \nabla\bullet\nabla C = 0$$

5.1.2 Nodal weighting operator

Terms such as the clustering or pairing reactions must be weighted by the volume of a node so that they can be appropriately added to a divergence operator. The weighting routine `volume` does the job.

5.1.3 Interface operator

Radiative or other surface fluxes must be weighted by an appropriate geometrical term, just as the bulk quantities are. The `interface` operator carries out this function.

5.1.4 Dirichlet operator

The `dirichlet` operator marks its solution variables as being fixed at the interfaces to which it is applied. A co-routine (phyterm) specifies what those fixed values should be.

5.1.5 Transient operator

Transient terms require a special implementation of the nodal weighting operator which can access historical values of its inputs.

5.1.6 Constraint operator

`PROPHET` allows the specification of constraints on the values of a field on either side of an interface. The most common case is where a quantity such as potential is desired to be continuous across an interface. In heterostructure device simulation, it is common to have a fixed ratio of the concentration of carriers on one side to the concentration on the other. The constraint relation may be more complex again, such as the ratio between the two interface quantities being some function of other variables. All such conditions are handled using the constraint operator.

5.2 Physical operators (“flux routines” or “phyterms”)

Associated with divergence operators	
`lapflux`	linear flux of n diffusing species $D\nabla C_n$
`equilflux`	flux of n diffusing species in the presence of electric field, with diffusivity depending on potential $D(\psi)(\nabla C_n + \xi_n C_n \nabla\psi)$ d
`diffusion`	generalized diffusivity $A\nabla B$
`drift`	drift operator $\mu A\nabla B$ - using central differences
`updrift`	drift operator $\mu A\nabla B$ - using upwinding
`coupleflux`	coupled diffusion flux $D(\psi)(X(\nabla B + \xi B\nabla\psi) + B\nabla X)$
`drift_diffusion`	Scharfetter-Gummel discretization of the drift-diffusion operator
Associated with nodal operators	
`two2one`	the chemical reaction A + B <-> C
`cluster`	the chemical reaction C + I <-> C
`poissonflux`	nodal charge, which adds all signed active dopants and electron/hole charge.
`elim_carrier`	calculate carriers based on potential
`set_active`	calculate electrically active concentration of n chemical species
`potflux`	similar to poissonflux, but for use in device simulation
`quasiFermi`	similar to elim_carrier, but for use in device simulation
`odefunc`	user function of n variables
`prod,add`	product,sum of N inputs
`divide`	first input divided by second
`copy, scale`	quantity divided by a scalar
`sqroot, exp, log, asinh`	square root, exponential, loarithm and arc-sinh of a quantity
Associated with the interface operator	
`segregation`	Segregation of n species between two materials $k(C_n[\mathrm{mat1}] - C_n[\mathrm{mat2}]/m)$
`radiation`	radiation from a boundary
`odesurf`	user function of n variables

Associated with the dirichlet operator	
`default_dirichlet`	Defines dirichlet boundary conditions
`device_dirichelt`	Defines electron,hole and potential concentration as a function of doping
Associated with the constraint operator	
`continuity`	Enforces continuity across an interface
Transient operators	
`ddt`	time derivative of a field
`addt`	product of a field with time derivative of another field
`psirhodot`	form $\psi q\left(\frac{\partial p}{\partial t} - \frac{\partial n}{\partial t}\right)$ for transient electrothermal simulation

The physical operators take fields or their gradients as input and form fluxes by combining them together in various ways. Some operators also use coefficients in the database, usually looked up by name based on the working region and the name of the input or output fields. If no coefficient is specified in the database or the user input file, the coefficient will default to zero except where otherwise specified.

In the pre-packaged diffusion operators, there is a convention that the electrical concentration of the species are distinguished from the chemical concentrations by the addition of an asterix to the chemical name, for instance `boron` and `boron*`. Such operators could of course be built up as combinations of the more primitive operators. The pre-packaged form gives convenient access to a fully formulated model.

5.2.1 lapflux

Inputs:	`[A,B,C,...]`	
Outputs:	`[X,Y,Z,...]`	
Coefficients	diffusivity	`library/physics/material/A/Dix`
Function	Computes the negative flux $D\nabla A$ for each of the inputs and stores it on the corresponding output. The number of inputs and outputs must be the same, and usually the outputs are the same as the inputs.The coefficient name is taken from the input field.	

5.2.2 equilflux

Inputs:	`[Sb*,As*,B*,P*,psi]`	
Outputs:	`[Sb,As,B,P]`	
Coefficients	neutral diffusivity	`library/physics/material/dopant/Dix`
	positive diffusivity	`library/physics/material/dopant/Dip`
	negative diffusivity	`library/physics/material/dopant/Dim`
	double negative diffusivity	`library/physics/material/dopant/Dimm`
	dopant sign	`library/physics/material/dopant/dsign`

Function	Computes the negative flux of N diffusing species in the presence of electric field, with diffusivity depending on potential $D(\psi)(\nabla C_n + \xi_n C_n \nabla\psi)$. The inputs are usually the electrically active dopant concentrations, followed by the potential.	

5.2.3 diffusion, drift and updrift

Inputs:	`[A,B]`	
Outputs:	`[X]`	
Coefficients	mobility	`library/physics/material/A/driftco.B` (defaults to 1.0)
Function	Computes the generalized negative diffusion flux $A\nabla B$ and stores it on the ouput X. The diffusion and drift operators are the same, the only difference being the output; when X=A the result is drift, when X=B the result is diffusion. The `updrift` operator uses the less accurate but more stable upwind differencing scheme.	

5.2.4 coupleflux

Inputs:	`[Xn,A1*,A2*,Am*,psi]`	
Outputs:	`[X,A1,A2,Am]`	

Coefficients	fractional diffusivity
	diffusivity components
	dopant sign
		`library/physics/material/A1/fraction.X`
		`library/physics/material/A1/{Dix,m,p,mm}`
		`library/physics/material/A1/dsign`

Function	Computes m coupled diffusion fluxes $f_X D(\psi)(X_n(\nabla A + \xi A \nabla\psi) + A\nabla X_n)$. The input X_n is the normalized defect concentration. The remaining inputs are the mobile dopant concentrations and potential. The diffusion flux is added to each of the dopant equations and to the defect equation.	

text

5.2.5 drift_diffusion

Inputs:	`[psi,electrons` or `holes]`	
Outputs:	`[electrons` or `holes]`	

Coefficients	carrier sign
	carrier diffusivity
	carrier mobility
		`library/physics/material/electrons/esign`
		`library/physics/material/electrons/Dix`
		`library/physics/material/electrons/mobility`

Function	Computes one carrier flux assuming constant mobility and diffusivity, by the Scharfetter-Gummel method.	

5.2.6 two2one

Inputs:	`[A,B,C]`	

Outputs:	[A,B,C]	
Coefficients	forward rate	library/physics/material/C/kf.two2one.A.B
	reverse rate	library/physics/material/C/kr.two2one.A.B
Function	computes the flux $k_f AB - k_r C$ and stores it with a negative sign on the A and B equations and a positive sign on the C equation	

5.2.7 cluster

Inputs:	[C,I]	
Outputs:	[C,I]	
Coefficients	forward rate	library/physics/material/C/kf.cluster.I
	reverse rate	library/physics/material/C/kr.cluster.I
		library/physics/material/C/background
Function	computes the flux $k_f CI - k_r(C - back)$ and stores it with a positive sign on the C equation and a negative sign on the I equation. The value back is the background concentration of the variable and serves to avoid numerical problems with driving C to 0.	

5.2.8 poissonflux

Inputs:	[electrons,holes,D1*,D2*,D3*]	
Outputs:	[psi]	
Coefficients	electron charge	library/physics/qcharg
	dopant sign	library/physics/material/D1/dsign
Function	computes the total charge at a node by adding carriers and dopants with appropriate signs. The inputs are the electrically active dopant concentrations. Used in process simulation where the dopants are available separately.	

5.2.9 elim_carrier

Inputs:	[psi]	
Outputs:	[electrons,holes]	
Coefficients	carrier sign	library/physics/material/carrier/esign
	intrinsic number	library/physics/material/ni
Function	computes the equilibrium carrier concentrations at a node as a function of the electrostatic potential $c = n_i \exp(-\xi\psi)$ where the potential is normalized and ξ is the carrier sign.	

5.2.10 set_active

Inputs:	[Sb,As,B,P]	

Outputs:	`[Sb*,As*,B*,P*]`	
Coefficients	solubility	`library/physics/material/B/Solubility`
	power law of solubility	`library/physics/material/B/m`
	coefficient for power law	`library/physics/material/B/beta`
Function	computes the electrically active concentrations from the chemical concentrations using either a hard solubility maximum or a soft saturation of the type $A + \beta A^m = C$. If both models are present for an impurity, the soft model takes precedence.	

5.2.11 potflux

Inputs:	`[electrons,holes,netdope]`	
Outputs:	`[psi]`	
Coefficients	electron charge	`library/physics/qcharg`
Function	computes the total charge at a node by adding carriers and net doping.	

5.2.12 quasiFermi

Inputs:	`[psi]`	
Outputs:	`[electrons,holes]`	
Coefficients	intrinsic number	`library/physics/material/ni-sze`
	carrier sign	`library/physics/material/carrier/esign`
	fixed quasi-Fermi value	`library/physics/material/carrier/qf.fixed`
Function	computes the carrier concentrations at a node assuming no transport. This is used to eliminate whichever carrier is not being solved in a single carrier or zero carrier device problem. $c = n_i \exp(-\xi(\psi - \phi))$ where the quasi-Fermi level ϕ is taken as zero if not specified. In the device modules, potential is not normalized.	

5.2.13 odefunc

Inputs:	`[c1,c2,c3,...]`	
Outputs:	`[c1,c2,c3,...]`	
Coefficients	user-specified	
Function	this is the template function for writing new nodal operators. Any set of reactions between any number of species can be inserted, and their coefficients retrieved from the input file using standard subroutines.	

5.2.14 prod,sum

Inputs:	`[c1,c2,c3,...]`	

Outputs:	`[X]`	
Coefficients	none	
Function	forms the product or sum of the specified inputs	

5.2.15 divide

Inputs:	`[A,B]`	
Outputs:	`[X]`	
Coefficients	none	
Function	X=A/B	

5.2.16 copy, scale

Inputs:	`[A]`	
Outputs:	`[X]`	
Coefficients	scale factor	`library/physics/material/A/Cstar`
Function	The copy operator transfers A to X. The scale operator divides A by the constant Cstar and stores in X	

5.2.17 sqrt,exp,log,asinh

Inputs:	[A]	
Outputs:	[X]	
Coefficients	none	
Function	The specified arithmetic function is performed on the field A and the results stored in X	

5.2.18 segregation

Inputs:	[A1,A2,A3,...]	
Outputs:	[A1,A2,A3,...]	

Coefficients:	segregation coefficient
	segregation rate
		`library/physics/mat1/A1/segregation-coeff.mat2`
		`library/physics/mat1/A1/segregation-rate.mat2`
		
		`library/physics/silicon/boron/segregation-coeff.germanium=3`
		means that the equilibrium germanium concentration is three times that in silicon.

Function	Computes the segregation flux $h(C_1 - C_2/m)$ between the concentrations of a dopant on the two sides of an interface.	

5.2.19 radiation

Inputs:	`[A1,A2,A3,...]`	
Outputs:	`[A1,A2,A3,...]`	
Coefficients:	radiation rate	`library/physics/mat1/A1/Krad`
	equilibrium concentration (defaults to 0)	`library/physics/mat1/A1/Cstar`
Function	Computes an outward flux $k(C - C^*)$ for each of the inputs and adds to the corresponding output equation.	

5.2.20 odesurf

Inputs:	`[c1,c2,c3,...]`	
Outputs:	`[c1,c2,c3,...]`	
Coefficients	user-specified	
Function	this is the template function for writing new surface operators. Any set of normal fluxes involving any number of species can be specified.	

5.2.21 default_dirichlet

Inputs:	none	
Outputs:	`[c1,c2,c3,...]`	
Coefficients	dirichlet value	`library/physics/mat1/ci/dirichlet.mat2`
Function	Provides the dirichlet value to be used for a field at a given interface. The region "*mat1*" will always be a real region, but "*mat2*" may be a boundary condition such as "**`exposed`**" or "**`cathode`**".	

5.2.22 device_dirichlet

Inputs:	netdope	
Outputs:	[psi,electrons,holes]	
Coefficients	dirichlet value	`library/physics/material/ni-sze`
Function	Computes the charge-neutral values of electrons,holes and potential at ohmic contacts as a function of doping.	

5.2.23 continuity

Inputs:	`c1`
Outputs:	`c1`

Coefficients	none
Function	Forces continuity of c1 across the interfaces to which the operator is applied

5.2.24 ddt

Inputs:	`[A,B,C]`
Outputs:	`[A,B,C]`
Coefficients	none
Function	Computes the time derivative of the fields and adds them to the corresponding equations

5.2.25 addt

Inputs:	`[A,B]`
Outputs:	`[X]`
Coefficients	none
Function	Adds $A\frac{\partial B}{\partial t}$ to equation X

5.2.26 psirhodot

Inputs:	`[psi,electrons,holes]`	
Outputs:	`[T]`	
Coefficients	electron charge	`library/physics/qcharg`
Function	form $\psi q\left(\frac{\partial p}{\partial t} - \frac{\partial n}{\partial t}\right)$ for transient electrothermal simulation	

6. Further examples

6.1 Using functions

This system solves

$$\boxed{\frac{\partial D}{\partial t} = \kappa\nabla^2(D^2)}$$

```
system name=test3a
+       sysvars=d
+       transient=d
+       tmpvars=d2
```

```
+       nterm=2
+       term0=box_div.lapflux(d2|d)@{silicon}
+       term1=transient.ddt(d|d)@{silicon}
+       nfunc=1
+       func0=prod(d,d|d2)@{silicon}

grid xloc=0,1

implant elem=d dose=1e14 range=0.5 sigma=0.05
dbase create name=library/physics/silicon/d/background rval=1e10
dbase create name=library/physics/silicon/d/scale rval=1e10
dbase createlist name=library/physics/silicon/d2
dbase create     name=library/physics/silicon/d2/Dix sval="1e-6/1e18"

dbase create name=options/timestep ival=1
dbase create name=options/movie sval=d
solve min=30 temper=1000 system=test3a
```

Note that the diffusivity is stored under d2, because that is the input of the lapflux.

6.2 Using arbitrary fields

Any field can be used in a function as long as it has been defined. This system solves

$$\frac{\partial D}{\partial t} = \kappa \nabla^2 \left(\frac{1}{D} \right)$$

using a field of "unity" as input to the divide operator.

```
system name=test3
+      sysvars=d
+      tmpvars=dtmp
+      transient=d
+      nterm=2
+      term0=box_div.lapflux(dtmp|d)@{silicon}
+      term1=transient.ddt(d|d)@{silicon}
+      nfunc=1
+      func0=divide(unity,d|dtmp)@{silicon}

dbase create name=library/physics/silicon/d/background rval=1e15
dbase create name=library/physics/silicon/d/scale rval=1e15

dbase createlist name=library/physics/silicon/dtmp
dbase create name=library/physics/silicon/dtmp/Dix sval=-1e30*1e-2

dbase createlist name=library/physics/silicon/unity
dbase create     name=library/physics/silicon/unity/class sval=permanent

grid xloc=0,1
implant elem=d dose=1e14 range=0.25 sigma=0.05
```

```
field set=unity val=1

dbase create name=options/timestep  ival=1
dbase create name=options/movie sval=d
solve min=30 temper=1000  system=test3
```

6.3 Using multipliers and dirichlet boundary conditions

This example solves

$$\nabla^2 D + 100D = 0$$
$$D(0) = 1$$
$$D(1) = 0$$

Since this is a steady state problem, no time is given.

```
system name=test4
+       sysvars=d
+       nterm=3
+       term0=box_div.lapflux(d|d)@{silicon}
+       term1=dirichlet.default_dirichlet(0|d)@{silicon/bulk,silicon/exposed}
+       term2=-100*volume.scale(d|d)@{silicon}

dbase create name=library/physics/silicon/d/background rval=0
dbase create name=library/physics/silicon/d/scale rval=1
dbase create name=library/physics/silicon/d/Dix rval=1
dbase create name=library/physics/silicon/d/dirichlet.exposed rval=1
dbase create name=library/physics/silicon/d/dirichlet.bulk rval=0

grid xloc=0,1

field set=d val=1

solve system=test4

graph elem=d log=0 ymin=-10 ymax=10
```

6.4 Diffusion with an arbitrary diffusivity in an arbitrary field

As an example with both drift and diffusion, consider

$$\frac{\partial B}{\partial t} = \nabla \bullet (D \nabla B + BD \nabla \psi)$$

The system can be written

```
system name=test6
+       sysvars=boron
+       tmpvars=bdco
+       nterm=3
+       term0=box_div.diffusion(dco,boron|boron)@{silicon}
+       term1=box_div.drift(bdco,psi|boron)@{silicon}
+       term2=transient.ddt(boron|boron)@{silicon}
+       nfunc=1
```

```
+       func0=prod(dco,boron|bdco)@{silicon}


grid xloc=0,1 elem=boron conc=1e15

implant elem=boron dose=1e14 energy=60

field   set=germanium val=1e20*exp(-0.5*(X-0.2+abs(X-0.2))/0.05)+1e16
field   set=psi val=-log(germanium/5e19)

# since we are evaluating dco outside the diffusion loop,
# must set temperature explicitly
dbase create name=options/dummy sval=1
dbase print name=options/dummy temper=1000

field   set=dco val="4*8.33e8/60*exp(-3.43/kT)*1e20/(germanium+1e19)"

dbase create name=options/movie sval=boron
dbase create name=options/timestep ival=1

solve min=30 temper=1000 system=test6
```

With the chosen sign of potential and drift term, the field opposes the boron diffusion. At the base of the germanium profile, the field goes to zero on the right but not the left, so that the driving force actually scoops boron up from the flat part of its profile and drives it uphill towards the germanium peak.

7. Developing new models

Often more complicated models will require reactions above and beyond the simple reactions listed above. Developing new modules to implement those reactions is intended to be easy. As an example, this is the C code that implements the `two2one` operator.

```
/*
 * Basic chemical reaction
 *
 * A + B -> C
 *
 *  $Header: /home/ulsi/prophet/CVS/platform/PDE/Fluxes/two2one.c,v 2.3 1997/07/20
01:13:22 conor Exp $
 */

#include "prophetc.h"
#include "grid.h"
#include "assemblist.h"
#include "pdeterms.h"
#include "mathpack.h"




/*-----------------t w o 2 o n e----------------------------------------
 * Implement
 * A + B -> C
 *
```

```
 * read arguments: [A,B,C]
 * write args:     [X,Y,Z]
 * normally X,Y,Z=A,B,C but not necessary
 * coefficients: library/physics/material/C/kf.two2one.A.B   forward coefficient
 * coefficients: library/physics/material/C/kr.two2one.A.B   reverse coefficient
 *----------------------------------------------------------------------*/



/*ARGSUSED*/
two2one( arglist )
   argdescrip
{
   real **sol_=0, **f_=0, ***df_=0;
   real kf, kr, val;
   int in;
   int rhs = ((*imtx)%10 == 1);
   int mtx = ((*imtx)/10 == 1);

   /* Sanity check */
   if( *nsol != 3 || *ndep != 3)
   {
      ndberr("two2one: bad args\n");
      fluxwhine( nsol, msol, ndep, mdep);
      return(-1);
   }

   switch( *path)
   {
   case FT_CONFIG:
      for( in = 0; in < *nsol * *ndep; in++) df[in] = 1;
      return(0);

   case FT_RUN:

      /*
       * Get the forward and reverse coefficients
       */
      kf = rscoeff3( "kf.two2one", mdep[2], mdep[0], mdep[1], *ireg);
      kr = rscoeff3( "kr.two2one", mdep[2], mdep[0], mdep[1], *ireg);

      /*
       * Convert the flat arrays into 2d arrays
       */
      sol_ = array2( *nsol, *nn, sol);
      f_   = array2( *nsol, *nn, f);
      df_  = array3( *ndep, *nsol, *nn, df);

      /*
       * Figure the reaction and its derivative
       */
      if( rhs)
        for( in = 0; in < *nn; in++) {
           val = kr*sol_[2][in] - kf*sol_[0][in]*sol_[1][in];
```

```
          f_[2][in] = val;
          f_[0][in] = -val;
          f_[1][in] = -val;
       }
     if( mtx)
       for( in = 0; in < *nn; in++) {
          df_[2][2][in] = kr;
          df_[0][2][in] = -kf*sol_[1][in];
          df_[1][2][in] = -kf*sol_[0][in];

          df_[2][1][in] = -df_[2][2][in];
          df_[0][1][in] = -df_[0][2][in];
          df_[1][1][in] = -df_[1][2][in];

          df_[2][0][in] = -df_[2][2][in];
          df_[0][0][in] = -df_[0][2][in];
          df_[1][0][in] = -df_[1][2][in];
       }

     /*
      * Uncovert arrays
      */
     (void)array2free( *nsol, *nn, sol_);
     (void)array2free( *nsol, *nn, f_);
     (void)array3free( *ndep, *nsol, *nn, df_);
     break;

     /*
      * ... Ignore all other calls.
      */
  default:
     break;
  }

  return(0);
}
```

The critical lines come just below the comment “Figure the reaction and its derivative”. The flux is set up as

```
kr*sol_[2][in] - kf*sol_[0][in]*sol_[1][in]
```

and stored with a positive sign in the pair equation and a negative sign in the phosphorus and interstitial equations. The sign convention comes from writing the total system as an equation with a zero right hand side:

$$\frac{\partial P}{\partial t} + (k_{pr}\Pi - k_{pf}PI) = 0$$

Right below that, the derivatives of the flux with respect to its inputs are calculated in the obvious way. The rest of the subroutine is essentially set-up for these few lines.

Starting from the top, the set of include files is basically standard. The comment at the start of the routine should always list what the expected input and output fields of this operator are, so that when someone else is assembling this operator into their property list they know what sequence to list the fields. It is also helpful to list the coefficients for your own or the next user's reference.

The subroutine gets a standard list of arguments **`arglist`** and their descriptions **`argdescrip`**.

int *path	whether to compute the flux (FT_RUN) or do set-up or clean-up
int *imtx	whether to compute the flux or its derivatives: 1=flux 10=derivative 11=both
int *ireg	index of region
int *nn	number of nodes to work on
int *dim	space dimension of operator
int *nsol	number of output variables
int *msol	indices of each output variables in the global list
int *ndep	number of input variables
int *mdep	indices of each input variable in the global list
real *coord	coordinates of points - for models which have an explicit spatial dependence (ick)
real *sol	the input variables, ordered with node index fast and variable index slow (ndep,nn)
real *gradsol	gradients of the inputs, for computing fluxes (nsol,dim,nn)
real *f	the output fluxes (nsol,dim,nn)
real *df	derivative of output fluxes with respect to inputs (ndep,nsol,dim,nn)
real *dgf	derivative of output fluxes with respect to input gradients (ndep,nsol,dim,dim,nn)

The argument list is set up so that the `phyterms` can be written in Fortran or C. For that reason, all integers are passed in as pointers to integers and the arrays are "flat", i.e. a single block of storage for each array, rather than a set of pointers to pointers to storage.

The first thing the subroutine does is check it got called with the right number of species.

The next thing is to check whether it is being called in configuration mode (path FT_CONFIG). The subroutine must return the coupling between its inputs and outputs. For this purpose the "`df`" array is overloaded. It is treated as an `nsol*ndep` array, each entry of which defines whether the output variable depends on the concentration(1) or gradient(2) of the input variable, or on both(3). The output index varies fastest, and the default dependence is none (0). In this case, all the outputs depend on all the inputs, so all values are set to 1.

Then it goes into its main sequence. No special setup or cleanup is required for this subroutine, so everything goes under the FT_RUN case. For peak performance, it is worthwhile precomputing the coefficients only once per timestep (FT_DT) and storing them to be used at FT_RUN, instead of recomputing them at every loop as done here. For just two coefficients the overhead is not very significant. The subroutine `rscoeff3` is a convenience routine which simply constructs the necessary string, looks it up in the library, and evaluates it at temperature. Here it is used to construct

```
library/physics/silicon/pipair/kf.two2one.phosphorus.interstitial
```

Note how the order of variables was specified in order to get this result, rather than, say,

```
library/physics/silicon/phosphorus/kf.two2one.pipair.interstitial
```

The code is written in the most general way so that any three species can be passed in; a simpler but less generic subroutine might be specific about the names and say instead

```
tcoeff("silicon/intcluster/kf.rate.interstitial")
```

Other helpful routines in this context are

`rscoeff2( name, is1, is2, *ireg)`	- get library/physics/material/var1/name.var2
`rscoeff( name, is, *ireg)`	- get library/physics/material/var1/name
`rcoeff( name, *ireg)`	- get library/physics/material/name
`tcoeff(name)`	- get library/physics/name

These all return `reals`.

The next three lines convert the flat arrays into their structured counterparts, to allow simple indexing. This allows the later use of df_[2][2][in] rather than having to compute offsets into the flat array.

The active lines, already discussed, follow. The order of derivatives is slightly counterintuitive; one might expect $\partial f_i / \partial s_j$ at node n to be `df[i][j][n]` but in fact it is stored as `df[j][i][n]`. In Fortran, the indices are more intuitive: `df(n,i,j)`.

Finally, after calculating the fluxes, the temporary pointer arrays are disposed and the subroutine returns 0. If some problem was encountered, it should return -1.

This template should serve as well for other, more complicated reactions. There is no limit on what goes into the C or Fortran subroutine carrying out this purpose, and modules containing thousands of lines of code have been easily integrated.

When writing a new module such as this, a common hazard is coding the derivatives incorrectly. The flag "`options/test.newton`" puts Prophet in a loop which repeatedly assembles the function at the starting condition, changes one variable, reassembles the system, and checks that the Jacobian correctly predicts the change. That is, it tests whether

$$[J]\begin{bmatrix}0\\0\\\varepsilon\\0\\0\end{bmatrix} = [R\{x+\varepsilon\}] - [R\{\varepsilon\}]$$

Any non-zero values on the left or right side of the equation are printed out and can be inspected for agreement.