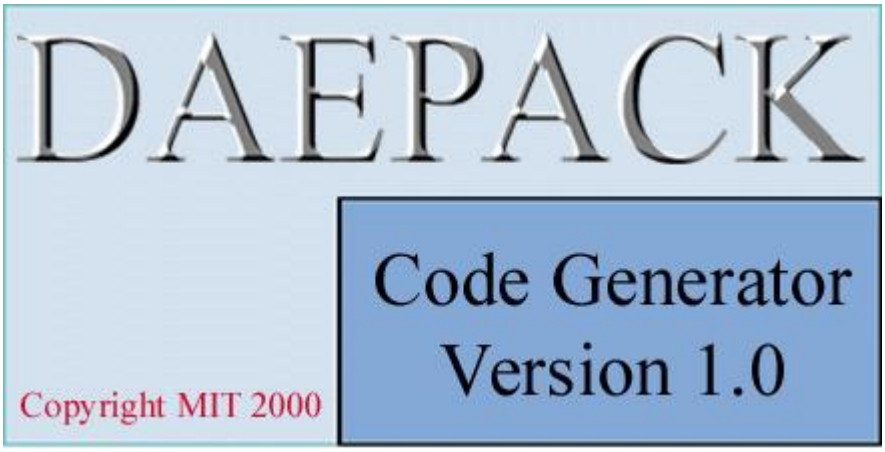
DAEPACK

Code Generation Manual
for Windows 9x and Windows NT



Overview

DAEPACK (pronounced D-A-E-PACK) is a symbolic and numeric library for performing general numerical calculations. What distinguishes DAEPACK from other libraries for numerical calculation is a set of symbolic components that operate directly on the user's Fortran model. For example, suppose the user has a general Fortran-90 model (in the form of a collection of one or more files containing one or more subroutines and functions) representing a system of equations of interest. This code may be an existing legacy model or a new model written for a particular application. DAEPACK takes this source code as input and generates a new set of Fortran-90 subroutines and functions that compute quantities such as general derivative matrices, sparsity patterns, and discontinuity-locked models. This new code can be compiled and linked to provide the information necessary when using the state-of-the-art numerical algorithms provided with DAEPACK or used with customized algorithms provided by the user.

The purpose of this manual is to describe how to use DAEPACK to generate automatically the information required when performing a wide variety of numerical calculations. DAEPACK is available on several platforms, including Windows 9x, Windows NT, UNIX (HPUX and Sun Solaris), and Linux. However, this manual describes how to use the Windows 9x and NT graphical user interface (GUI) provided with DAEPACK Version 1.0. A description of the command-line interface (CLI) available with DAEPACK on other platforms is provided in the Appendix C. GUIs will be available for these other platforms in later releases. As mentioned above, DAEPACK provides both symbolic *and* numeric components, however, a detailed description of the numeric components is provided in other manuals (references to more information are provided in the documentation following).

Outline

1. Introduction

The importance of modeling and simulation is of little dispute in nearly every discipline of science and engineering. Detailed simulations provide valuable insights into the behavior of existing processes and the design of new processes. It is assumed that the reader of this manual is well aware of the need for performing numerical calculations and the difficulty in performing them correctly. It is further assumed that the reader has performed numerical calculations by detailed programming in Fortran. Although this manual does not describe how to perform numerical calculations, it does describe how most of the additional information (beyond simply the residual evaluator) required when performing such calculations using state-of-the-art algorithms can be obtained automatically.

The layout of this manual is as follows. First, a description of the DAEPACK code generation GUI available for Windows 9x and Windows NT is presented. The user is taken through the steps of how to create model files, translate the model files so that DAEPACK can analyze them, and generate new code automatically for performing tasks such as analytical derivative evaluation, obtaining sparsity patterns, and performing hybrid discrete/continuous simulations. This is followed by a description of the automatically generated code and a small illustrative example (Appendices A and B). A description of a command-line interface is then given (Appendix C). Finally, a discussion about some subtle aspects of how array variables must be treated in DAEPACK is presented (Appendix D).

2. Graphical User Interface

On the Windows 9x and Windows NT platforms, the code generation components of DAEPACK are accessed through a graphical user interface (GUI). This section describes how to access these components and automatically generate new code from existing Fortran-90 models in order to calculate quantities such as analytical derivative matrices, sparsity patterns, and discontinuity-locked models. A description of the automatically generated code is provided in Appendix A.

Although this manual describes how to use the GUI on the Windows 9x and Windows NT platform, the generated code is as platform independent as the original source code. The code can be generated on a PC running Windows and moved to another platform to be compiled and linked into a larger application. The generated code uses a variety of supporting routines for performing the necessary calculations. These supporting routines are contained in a library that must be linked into the application using the generated code. Of course, the library it platform dependent. More information about using the generated code is provided in Appendix A.

2.1 Main Window

The DAEPACK Code Generation GUI can be accessed like most other Windows applications, either through the Start menu or by clicking on the DAEPACK icon in the DAEPACK folder. When the application is started, the main window shown in Figure 1 appears. The functionality of DAEPACK is accessed through the menu bar at the top of this main window. A status bar appears at the bottom of the main window, describing the current status of the translation and code generation process. Each item contained in the main menu is described below.

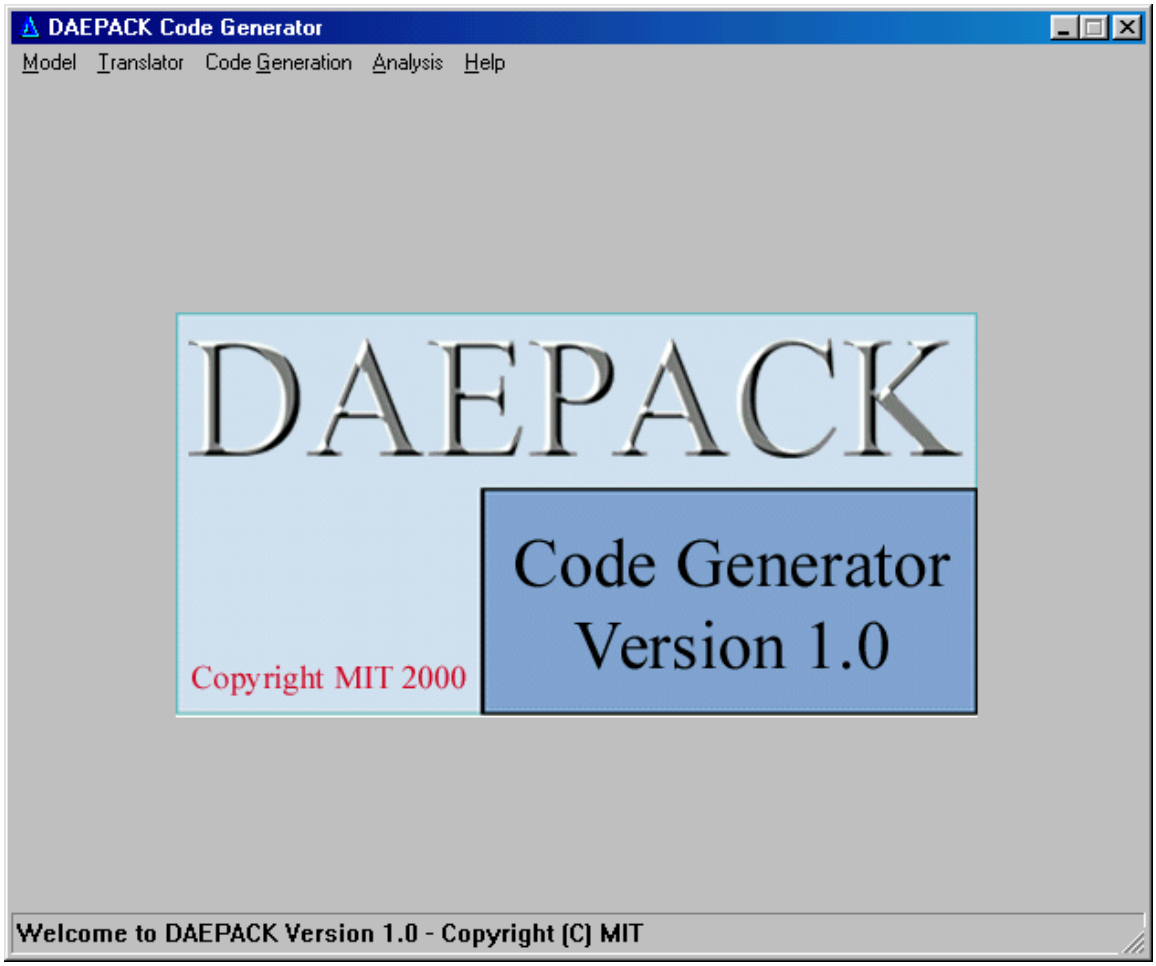


Figure 1. DAEPACK Code Generation GUI main window.

2.2 Model Options

In DAEPACK, a problem of interest is stored as a *model file*. A model file, which typically has a .mdl extension, is simply a list of one or more names and locations of Fortran source files that define the model. Since the Fortran model of interest may depend on several source files, the model file is simply a convenience. The user specifies all the files defining a particular problem once and every time the model is analyzed by DAEPACK, the model file keeps track of the source file dependencies and location.

The following options are available through the **Model** menu item:

- **Working Directory**
- **Create Model**
- **Open Model**
- **Display Model**
- **Save**
- **Save As**
- **Print**
- **Close**
- **Exit**

The last two options (**Close** and **Exit**) are self-explanatory: **Close** removes the current model from memory without exiting the application and **Exit** terminates the application. The other options require further discussion.

2.2.1 Changing the Working Directory

The **Working Directory** item in the **Model** menu allows the user to change the directory from which files will be searched when specifying a new model or opening an existing model. Creating and opening models will be described in more detail in sections 2.2.2 and 2.2.3, respectively. In addition, the working directory specified is where generated code will be saved. The **Working Directory** dialog is shown in Figure 2. In this dialog, you simply specify the folder you wish to be the working directory.

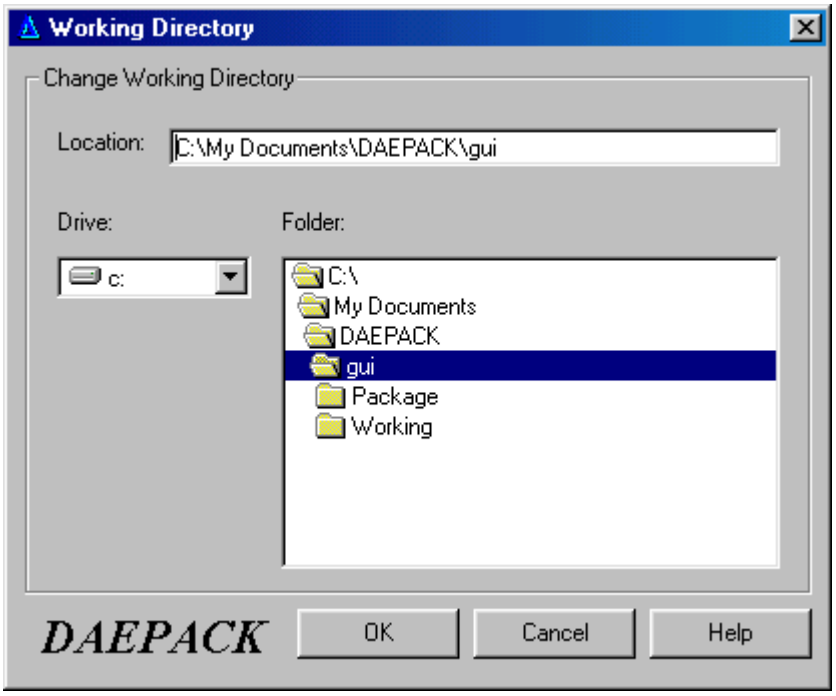


Figure 2. Working Directory dialog. The user can use this dialog to change the directory from which files are searched and where generated files are saved.

2.2.2 Creating New Models

As stated above, a model file must be specified in order to use the code generation components of DAEPACK. Clicking on the **Create Model…** option in the **Model** menu will access the model creation dialog shown in Figure 3.

In this dialog, specify a model name and create a list of source files by either typing the name and location in the 'Source file to add' text box or click on the appropriate file in the file list box on the left side of the dialog. Files are added by clicking on the 'Add File' button when a file name appears in the 'Source file to add' text box. In order to remove a file from the model, click on the file to remove in the current source file list and press the 'Remove File' button. This dialog also allows the user to specify the source file language, however, in version 1.0 of DAEPACK, only Fortran is permitted. Future implementations will support a variety of input languages (e.g., heterogeneous models formulated as a set of Fortran and C files). Once the model has been completely specified (e.g., all pertinent source files and their location have been

added), click on the 'OK' button at the bottom of the dialog. Upon completion, the status bar at the bottom of the main window will indicate the name of the model currently loaded.

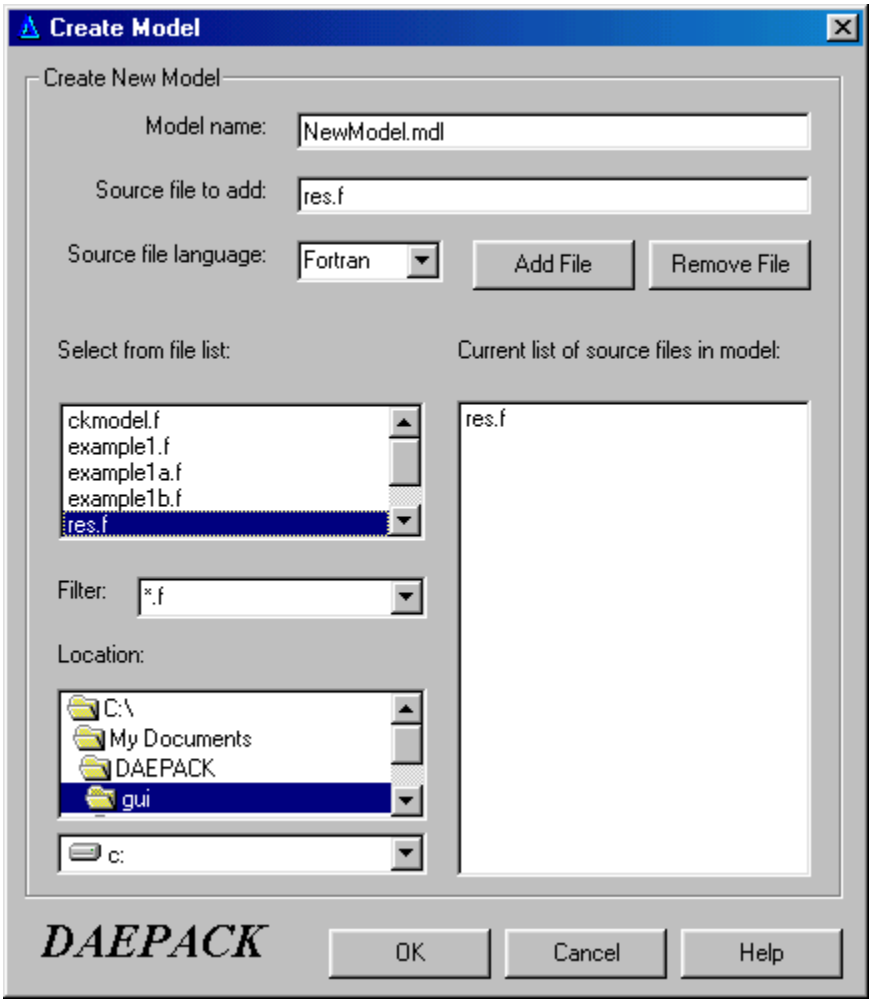


Figure 3. Create Model dialog. New models are created within this dialog by specifying a model name and list of source files.

2.2.3 Opening Existing Models

The section above describes how to specify new models. Once a model has been created and saved, it can be reloaded during a subsequent DAEPACK session by clicking on the **Open Model…** item of the **Model** menu. Note that when the model is first created the list of source files and their location is specified. These source files must be in the same location when the model is reopened. Figure 4 contains the dialog used to open existing models.

The list of available models in the working directory (described in section 2.2.1) appears on the right of this dialog box. The filter box below this list can be changed if the model file has an extension other than .mdl. The current directory can be changed through the 'Location' list on the left of this dialog. Once the model has been specified (either by typing in the name in the 'Model Name' text box or clicking on the appropriate file in the list box), click 'OK'.

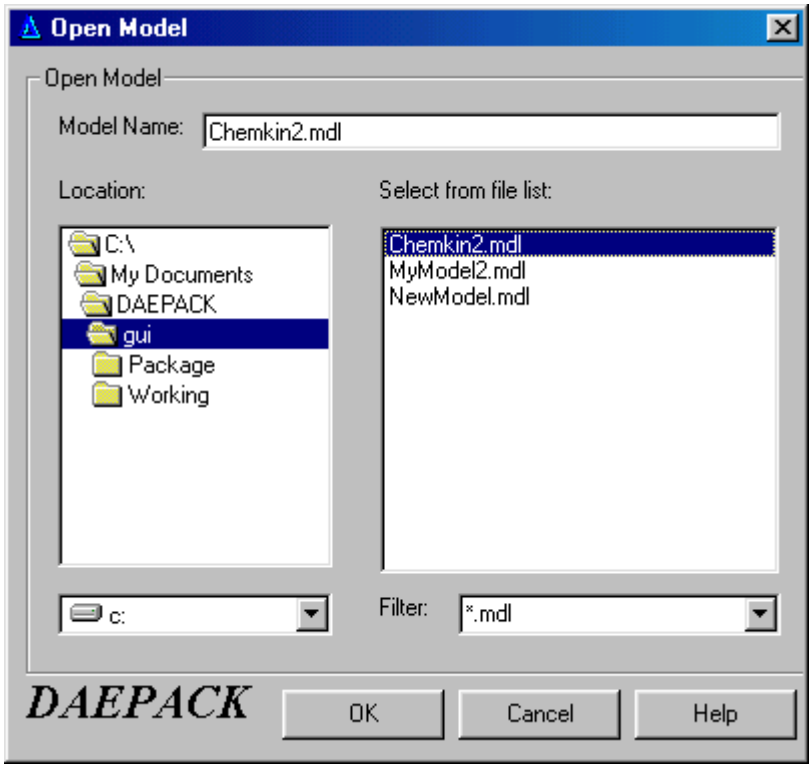


Figure 4. Open Model dialog. Existing models are loaded into memory within this dialog.

2.2.4 Displaying Models

The model currently loaded into memory can be examined (i.e., examine the list of files contained in the current model file) by clicking on the **Display Model…** item in the **Model** menu. A simple dialog box appears containing a list of the model's source files and their location.

2.2.5 Saving and Printing Files

Clicking the **Save…** or **Save As…** items in the **Model** menu will save the model currently loaded. Using the **Save As…** option allows the model name to be changed from its original specification. The **Print…** item in the **Model** menu simply prints out a list of the model's files. The dialog box for these options should be self-explanatory thus figures are omitted here.

2.3 Translation

Once a model has been loaded into memory (either a new model created or an existing model opened), the next step is to translate this model so that DAEPACK can analyze it. Clicking on the **Translate Model…** item in the **Translator** menu performs this task. Figure 5 contains the Translation dialog.

Click on the 'Start' button to begin file translation. (A model must be loaded into memory before this button becomes active.) Diagnostic and information messages will be sent to the window during translation. The 'Abort' button can be used to terminate the translation process. Once the model has been translated, press the 'Dismiss' button to remove this dialog box. The status of the translation process will appear at the top of the Translation dialog and in the status bar on the main menu. Before any code is generated (see below), all of the source files contained in the model file must have been successfully translated.

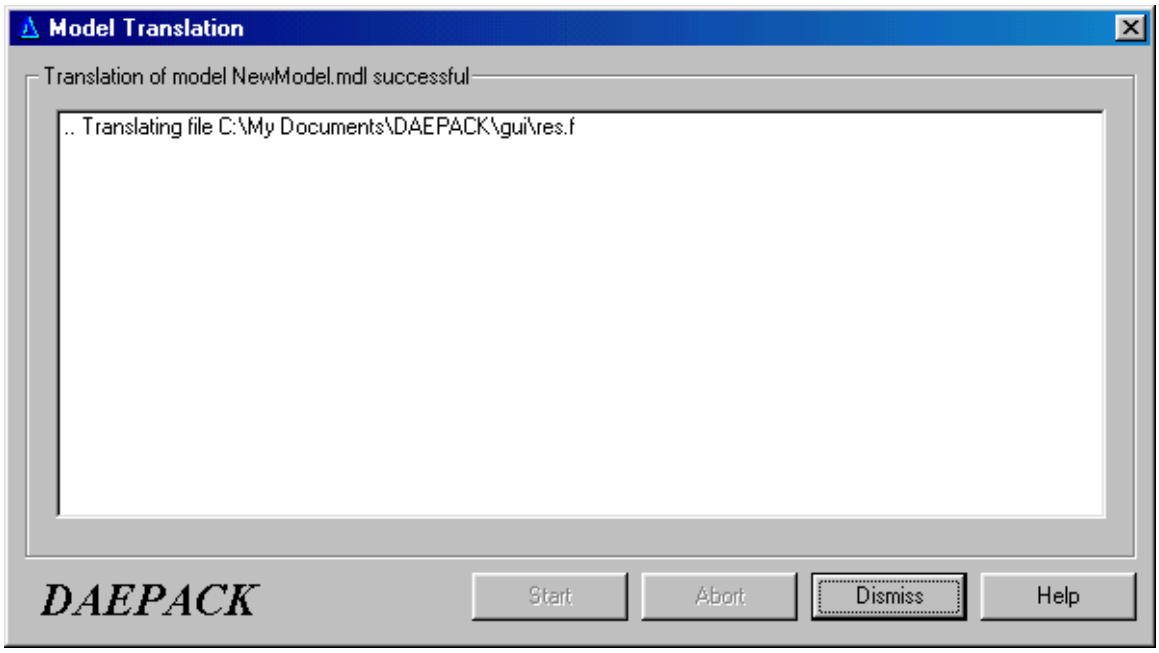


Figure 5. Translation dialog box.

2.4 Code Generation Options

As stated above, once the source files have been successfully translated, DAEPACK can analyze the model in order to generate useful information. Clicking on the **Generate Code…** item in the **Code Generation** menu brings up the code generation options form. Currently, three options are available: 1) analytical derivative matrices, 2) sparsity patterns, and 3) discontinuity-locked models. Future releases of DAEPACK will support a wider variety of code generation options, including interval extensions of general systems of equation and code generation templates for generating code for specific numerical routines (including the DAEPACK numerical components, DVODE, DASSL, and several others). The options for the currently supported code generation are described below.

2.4.1 Analytical Derivative Matrices

Clicking on the **Generate Code…** item in the **Code Generation** menu brings up the dialog shown in Figure 6. The desired code to generate can be selected by pressing on the appropriate tab in this dialog.

First, consider the derivative matrix options shown in Figure 6. The first field to specify in this options box is the name of the root program unit. DAEPACK assumes that every model evaluation, no matter how many program units (subroutines, functions, etc.) contained in the model, will have one entry point. This is the program unit (e.g., subroutine) that is called to evaluate the model. A list of possible program units can be found by pressing on the button to the right of the 'Root program unit' text box. In order for DAEPACK to generate code it must know the dimensions of each of the active variables (i.e., variables that depend directly or indirectly on the independent variables described below). Because of this, the user must specify the dimensions of all active variable arrays in the root program unit (this information is propagated to program units below the root program unit so the user need not specify these). DAEPACK will inform the user if there are unspecified variables. This should not place too much of a restriction on the user. If an array is of dimension `N` and `N` is in the argument list of the root program unit, then the array can be

dimensioned with this variable (rather than its numeric value). If an array is used as workspace then it must be dimensioned to a value (symbolic or numeric) greater than or equal to the largest index of any variable used. This is described in more detail in Appendix D.

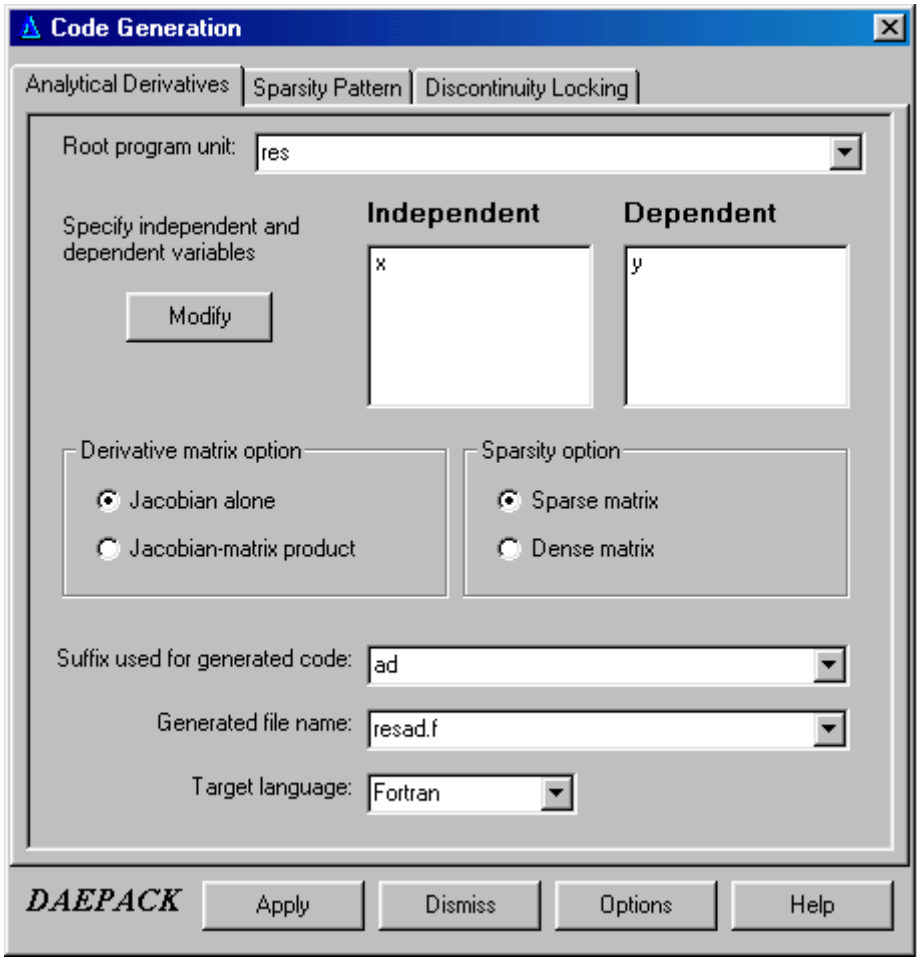


Figure 6. Analytical derivative matrix code generation options.

The next entries to specify are the independent and dependent variables. These may be specified in two ways, either as preprocessor directives in the source code or through the GUI. The independent and dependent variables can be specified in the source code directly by inserting the following DAEPACK preprocessor directives in the source code of the root program unit:

```
!!independent { comma-separated-variable-list }
!!dependent   { comma-separated-variable-list }
```

These directives can appear anywhere in the root program unit source but must begin in column 1. The variables specified in the comma-separated list can be any of the variables declared in the root program unit (including arguments, local variables, and common block variables). Each variable must be a floating-point type (e.g., real, double precision, etc.). The variables can be scalars, arrays, array entries, or array slices. An array variable appearing in the list without subscripts will be taken as full array. If a full array is specified as an independent or dependent variable, then the corresponding array variables must be dimensioned to their exact value (symbolic or numeric). The order of the independent and dependent variables in these lists is the ordering that will be used in the resulting Jacobian matrix. For example, the preprocessor directives:

```
!!independent { x(5), x(3), t }
```

```
!!dependent   { w, y, z }
```

will result in the following Jacobian matrix:

$$J(x(5), x(3), t) = \begin{pmatrix} \frac{\partial w}{\partial x(5)} & \frac{\partial w}{\partial x(3)} & \frac{\partial w}{\partial t} \\ \frac{\partial y}{\partial x(5)} & \frac{\partial y}{\partial x(3)} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial x(5)} & \frac{\partial z}{\partial x(3)} & \frac{\partial z}{\partial t} \end{pmatrix}$$

where `t`, `w`, `y`, and `z` are scalars and `x` is an array of dimension greater than five. If the preprocessor directive does not appear in the source code or you wish to modify these variables (e.g., change the variables or their order), press the 'Modify' button on the options form. This will bring up the variable modification dialog shown in Figure 7.

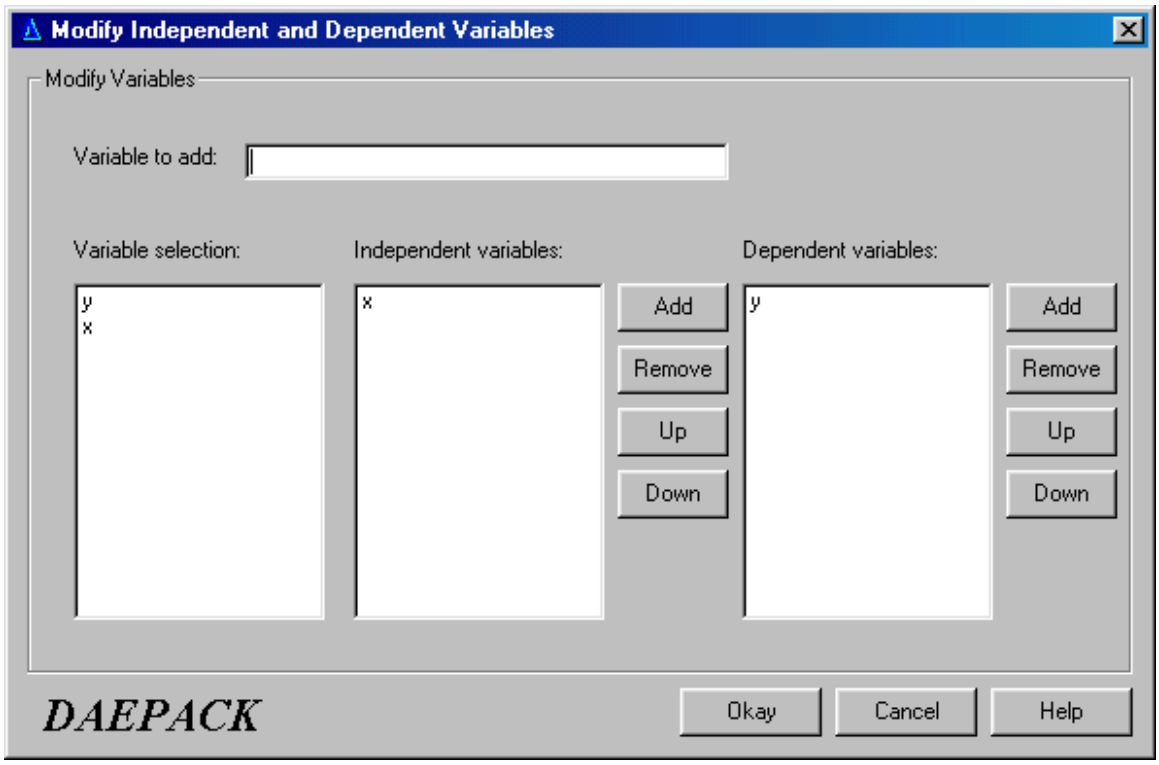


Figure 7. Modify Independent and Dependents dialog. This dialog is used to specify independent and dependent variables and their order used in the generated derivative code.

The list box on the far left of this dialog contains the set of variables that may be set as independent and/or dependent. In order to add a variable, click on the variable name in the 'Variable selection' list box or type the name in the 'Variable to add' text box. Press the appropriate 'Add' button to select this as an independent or dependent variable. To remove a variable, click on the variable in the independent or dependent variable list and click the appropriate 'Remove' button. To change the position of the variable (and, thus, the order in the resulting Jacobian matrix), click on the variable in the independent or dependent variable list and press the 'Up' or 'Down' button. Click the 'Okay' button to accept the change and return to the code generation options dialog. Press 'Cancel' to return without accepting the change.

Once the independent and dependent variables have been specified, additional derivative code generation options may be set. Currently, two derivative code generation options are available, the derivative matrix and storage options. By selecting 'Jacobian-matrix product', code is generated that returns the following quantity:

$$J(x)S$$

where $J(x)$ is the normal Jacobian matrix (partial derivatives of the dependent variables with respect to the independent variables) and S is an arbitrary conformable matrix that can be specified by the user. Selecting 'Jacobian alone' results in generated code that simply returns the Jacobian matrix. In addition to the derivative matrix options, two derivative matrix storage formats may be specified, sparse or dense. If 'Sparse matrix' is selected then the Jacobian matrix is returned in coordinate format, that is, a real array containing the nonzero matrix entries and two integer arrays containing the row and column indices of the sparse matrix. The actual values contained in the real array may or may not be zero. The values returned are based on the sparsity pattern of the Jacobian, not their current numeric value. Selecting 'Dense matrix' results in generated code that returns a two-dimensional array containing the full derivative matrix. The three remaining entries that must be filled in this option form are the suffix appended to the end of generated program units, name of the generated file, and target language. All of the program units contained in the original model will be stored in a single file with name specified in the 'Target file name' text box. The final option, 'Target language', allows the user to specify what type of code is to be generated (C, Fortran, etc.). Currently, this option is locked at Fortran, however, future releases will allow code to be generated in different languages (when possible). Note that the 'Suffix used for generated code' and the 'Generated file name' will be given default values once the root program unit is specified, however, the user is free to set these to any valid names (i.e., suffixes resulting in valid Fortran names and valid file names). Click the 'Apply' button to begin the code generation process. Before any code is generated, a form will pop up if there are any arrays contained in the argument list. This new form will display the specified dimension of these arrays and ask whether or not they are acceptable (see Appendix D). If they are acceptable, click 'Accept' to close this form and click 'Apply' (again) on the code generation options form to generate code. If these dimensions are not acceptable, press 'Reject', dismiss the code generation options form, and close the current model. Re-load and re-translate the model once the array dimensions have been set properly. Appendix D contains a more detailed description of how array arguments must be dimensioned when using DAEPACK. After dismissing the variable validation form, clicking the 'Apply' button will bring up the code generation output window shown in Figure 8 and begin the code generation process. Clicking the 'Dismiss' button will close the options dialog without performing any code generation. The 'Options' button can be used to specify additional code generation options. Finally, the 'Help' button provides some additional help specific to the code generation process.

The code generation output window (shown in Figure 8) displays diagnostic and information messages during code generation. The generated file will be stored in the working directory (see Section 2.2.1) under the name specified in the options form. A description of the generated code is contained in Appendix A.

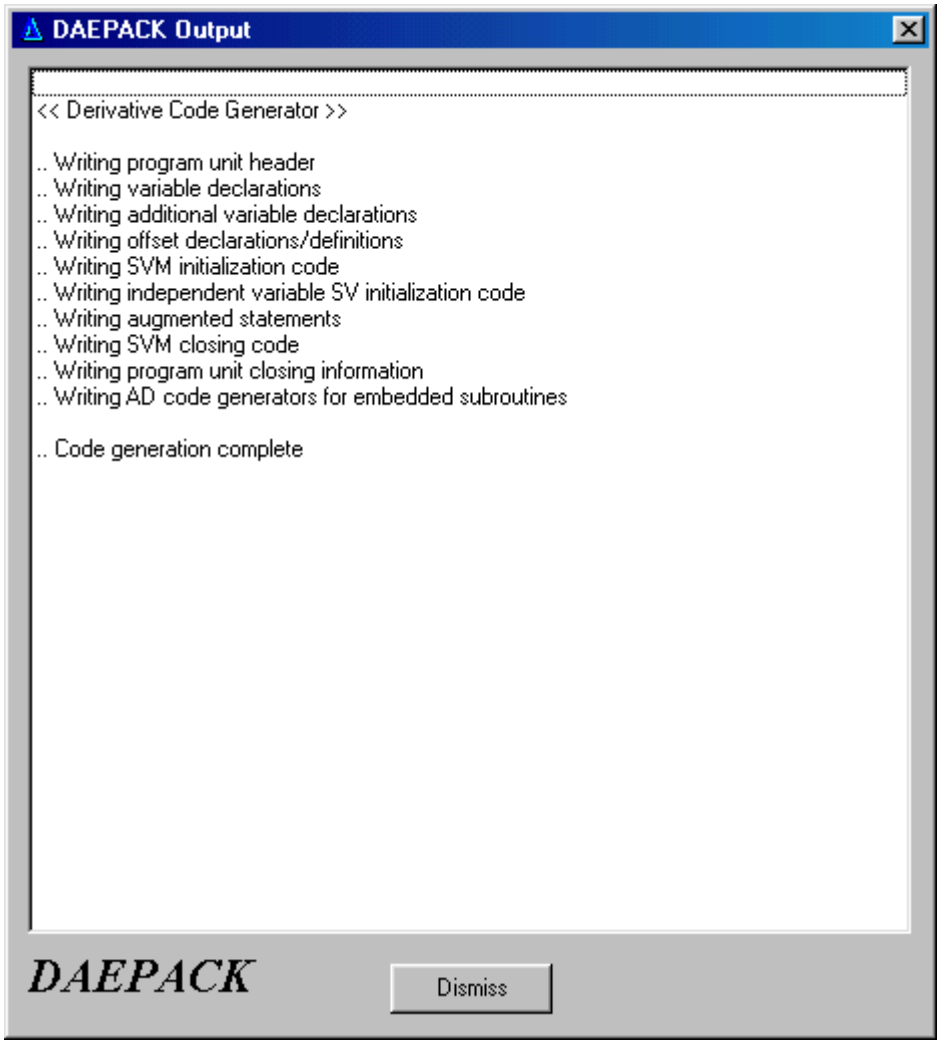


Figure 8. Code generation output window.

2.4.2 Sparsity Patterns

The sparsity pattern of a system of equations has numerous applications, many of which are performed by various DAEPACK numerical components. For example, the sparsity pattern can be used prior to numerical calculation to determine if the problem is structurally consistent, well-posed, or structurally high index. Graph algorithms can be applied to the sparsity pattern to identify an equation and variable permutation that can be exploited during numerical calculation. For example, if we are trying to solve a large, sparse set of nonlinear algebraic equations and the corresponding incidence matrix (i.e., sparsity pattern) is reducible, then it is possible to permute the matrix into block lower triangular form. The entire system can then be solved by sequentially solving the smaller diagonal blocks. This approach, implemented in a DAEPACK numerical component, has proven to be very efficient and robust compared to solving the entire system simultaneously. Furthermore, with the sparsity pattern, sparse linear algebra packages can be employed during the numerical calculation, often dramatically increasing computational efficiency and reducing memory requirements. This section describes how to generate a new set of subroutines and functions that will determine the sparsity pattern of the original model.

Clicking on the Sparsity Pattern tab of the **Code Generation** dialog box reveals the sparsity pattern options (see Figure 9).

The options for generating sparsity pattern code are similar to those for the derivative code generation. The user must specify a root program unit (the procedure called to perform a model evaluation), a set of independent and dependent variables, a suffix used for generated program unit names, the name of the file being generated, and a target language. Like the derivative matrix generation, the order of the independent and dependent variables appear in the list boxes in this dialog determine their numbering in the computed sparsity pattern. As described in the previous section, the independent and dependent variables can be specified either through preprocessor directives or by pressing the 'Modify' button in the sparsity pattern options box. The same restriction on array arguments of the root program unit described in the analytical derivative section also applies here: all active variable array arguments must have all of their dimensions set (either symbolically or numerically). Again, the 'Target language' option is locked to Fortran. Upon clicking 'Apply', the code generation output window will display diagnostic and information messages during code generation. The generated file will be stored in the working directory (see Section 2.2.1) under the name specified in the options form. A description of the generated code is contained in Appendix A.

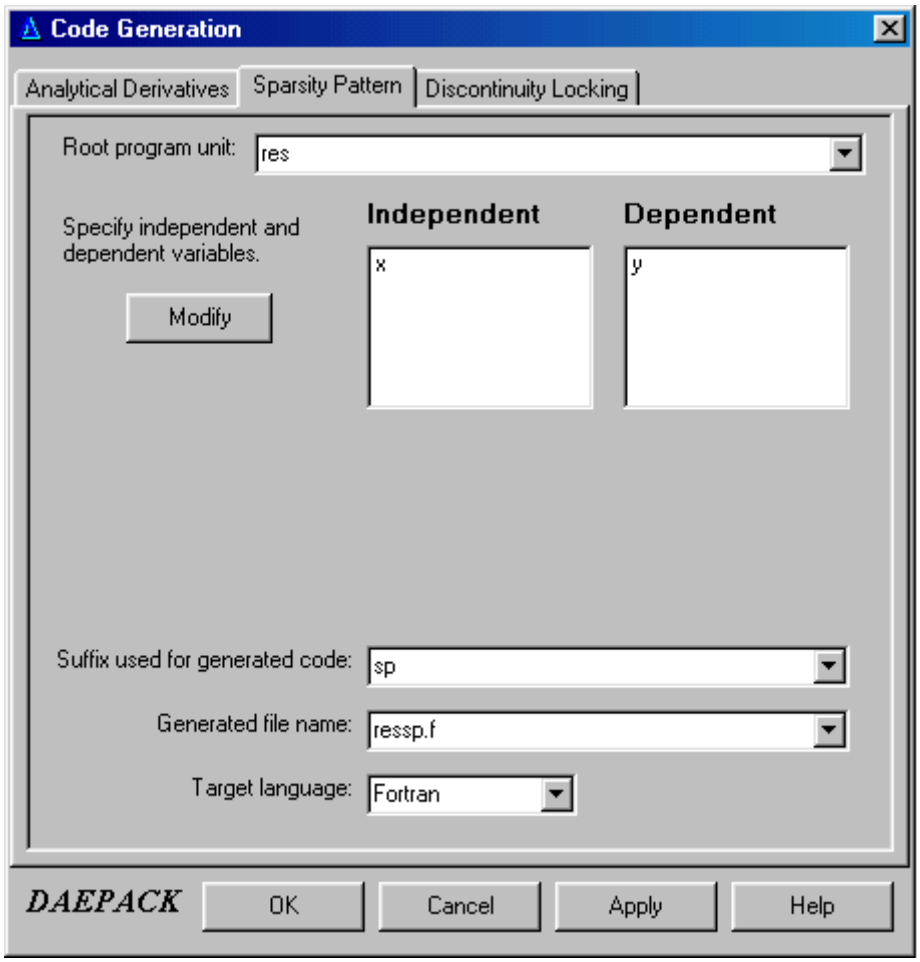


Figure 9. Sparsity pattern code generation options.

2.4.3 Discontinuity-Locked Models

A pervasive problem in the numerical solution of mathematical models is the presence of discontinuities. Within the model code, these discontinuities take the form of look-up tables and intrinsic functions such as MIN, MAX, and SGN in addition to the obvious IF equations. These discontinuities may appear for a variety of reasons. For example, discontinuities are introduced when modeling discrete aspects of a continuous system, such as the closing of a valve in a chemical process model or operation of a diode within an electrical circuit. Discontinuities also arise due to modeling abstractions. For example, physical

property relationships are often represented as piecewise continuous semi-empirical relationships valid over certain ranges of composition, temperature, and pressure. The discrete points in time during a numerical integration that a discontinuity occurs are referred to as *events* in the simulation literature. Events come in two main varieties*: time events* and *state events*. A time event is a discontinuity that occurs at a specified value of the integration variable and is thus known a priori. A state event occurs when a combination of one or more state variables satisfies some condition (referred to as a *state condition*). Obviously, the time of occurrence of a state condition will, in general, not be known a priori but must be determined during the course of a dynamic simulation. A model containing discontinuities is sometimes referred to as a hybrid discrete/continuous model and can be represented in a *hybrid automaton* framework. This is a continuous time formulation where the continuous aspects of the model are represented by a collection of continuous DAEs (or ODEs). The discrete nature of the problem defines a set of *modes* of the hybrid automaton each characterized by a distinct DAE. Accompanying the DAE in each mode is a mapping from the set of discrete states to the set of modes and an associated *state transfer function* that defines the initial conditions in the new mode.

Although discontinuities are common, they do present some problems when performing numerical calculations if not handled properly (we will refer to discontinuities that are not explicitly handled as *hidden discontinuities*). It is well known that the direct application of standard integration codes to a model containing hidden discontinuities can be inefficient, cause integration failures, and, in the worst case, lead to incorrect results that may escape the modeler's attention. Although less widely known, the situation is much worse when performing parametric sensitivity analysis on a model containing hidden discontinuities. In this case, the computed sensitivity trajectories will most certainly be quantitatively and qualitatively incorrect.

Several approaches have been developed to address the numerical problems associated with hidden discontinuities. At one extreme is the modification of the numerical integrator to detect and handle the discontinuities. Alternatively, one can replace all discontinuous aspects in the model with smooth approximations. The first approach has the following disadvantages: 1) it requires the modeler to use a numerical integrator specially modified for discontinuity handling and this type of integrator may not be best suited for integrating the dynamic model of interest and 2) the modeler has limited control over the discontinuity handling process. The second alternative, replacing discontinuities with smooth approximations, is not always suitable because it requires the user to search for all hidden discontinuities and replace them with an appropriate smooth approximation. Furthermore, the smooth approximation may not capture the dynamic behavior of interest (which may not be immediately obvious). A third approach for dealing with these numerical difficulties is to explicitly handle the discontinuities. In this approach, the modeler must *uncover* all of the discontinuities embedded within a hybrid discrete/continuous model and construct the corresponding *discontinuity functions*. A discontinuity function is a real-valued function with the following property: a discontinuity occurs when one or more discontinuity functions cross zero. For example, suppose the following state condition is embedded within a model:

```
IF(X(I) > 10.0 .AND. X(2*I) < 0.0) THEN
   .
   .
   .
END IF
```

The discontinuity functions corresponding to this state condition are $g_1(x)=x_i-10.0$ and $g_2(x)=-x_{2i}$. Obviously, a necessary condition for the state condition above to activate or deactivate is g_1 or g_2 crossing zero. In addition to extracting the discontinuity functions from the model, the modeler must be able to perform a *locked* model evaluation. That is, the modeler must be able to lock the model in a particular mode regardless of the mode actually implied by the current values of state variables. Given the discontinuity functions and the ability to lock the model into a particular mode, sophisticated event location algorithms can be used to perform the hybrid discrete/continuous calculation efficiently, robustly, and correctly. Requiring the modeler to extract all of the information described above is quite a formidable task, making the other two approaches described above more attractive. However, DAEPACK can generate automatically new code that extracts all of the hidden discontinuities and allows the modeler to lock and unlock the model evaluations. A description of the numerical algorithms using this generated code is beyond the scope of this manual. However, detailed information about how to use the generated

code with the DAEPACK component DSL48E for hybrid discrete/continuous simulation as well as a description of how to use the generated code with custom event handling algorithms is described in the DAEPACK DSL48E manual.

The options available for generating a discontinuity-locked model are shown in Figure 10. All the user must specify is the root program unit, the suffix for generated code, the name of the generated file, and the target language (again, locked to Fortran). All of these options have been described in the previous two sections. A description of the generated code is contained in Appendix A. As described above, a detailed description of how to use this code is contained in the DAEPACK DSL48E manual.

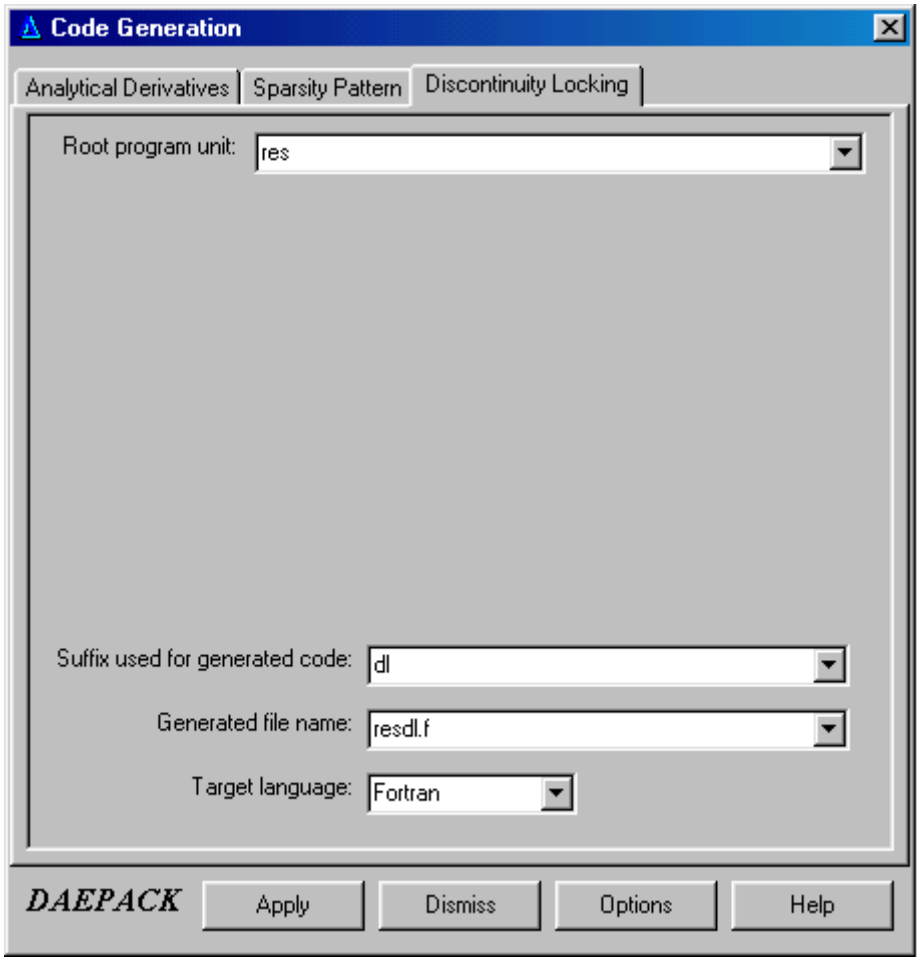


Figure 10. Discontinuity-locked model code generation options.

3. Conclusion

This manual describes how the code generation components of DAEPACK are accessed through the GUI available on the Windows 9x and Windows NT platform. The command-line version available with DAEPACK on other platforms is described in Appendix C. A description of the generated code is contained in Appendix A and an illustrative example is outlined in Appendix B.

This manual describes how to generate code for performing useful tasks such as computation of analytical derivative matrices, sparsity patterns, and discontinuity-locked models. A description of how these new codes are used during a numerical calculation is provided with the DAEPACK numerical component

manuals. For example, the analytical derivatives and sparsity pattern are used during the solution of large, sparse sets of nonlinear algebraic equations using the DAEPACK block solver component and numerical integration and parametric sensitivity analysis using the DAEPACK DSL48S component. The discontinuity-locked model is used by the DAEPACK DSL48E component for hybrid discrete/continuous simulation and parametric sensitivity calculation.

Although, only three types of generated code are described here, the underlying ideas and techniques used in the DAEPACK code generation components are very general and can be used to generate automatically additional useful information. For example, future implementations of DAEPACK will provide a code generation component that allows the user to construct new code for computing *interval extensions* of the original model.

Appendix A. Description of Generated Code

This appendix describes the code generated automatically by DAEPACK. As described in this manual, DAEPACK assumes that every model evaluation, no matter how complex, will have a single entry point. This may be a subroutine or function that is called to perform the model evaluation and return the results. This single program unit is referred to as the root program unit. The GUI allows the user to specify how the generated code will be named (by specifying a suffix that will be appended to generated program unit names). The program unit with the same name as the root program unit and suffix specified by the user is what is called to perform a model evaluation and compute the additional information (e.g., analytical derivatives or sparsity patterns). This new root program unit will have the same argument list as the original but augmented with additional arguments that return the desired information. The remainder of this appendix will describe how the argument list is modified and what the user must specify and how the results are returned.

The example used here is a small DAE model adapted from Birta *et al*. This example is kept purposefully small to illustrate the ideas, not the power of DAEPACK. The source code for the original model is shown below.

```
C ============================================================
C   Model of Example 4 of Birta et. al(1985)
C   Ref : L.G.Birta, T.I.Oren, and D.L.Kettenis, "A Robust
C         Procedure for Discontinuity Handling in Continuous
C         System Simulation", Tran. Soc. Computer Simulation,
C         Vol.2, No.3, p.189, 1985
C ============================================================
      subroutine res(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar)
!!independent{ w,wdot }
!!dependent  { delta  }
      implicit none
      integer neq,ires,ichvar,ipar(1)
      double precision t,w(neq),wdot(neq),delta(neq),rpar(1)
      double precision x1,x2,x3,x1dot,x2dot,v1,v2,v3,z3,z4
      double precision a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12
      parameter(a1=12.0d0,a2=0.24d0,a3=12.0d0,a4=0.24d0)
      parameter(a5=13.64d0,a6=-11.64d0,a7=-50.0d0,a8=0.0d0)
      parameter(a9=-11.64d0,a10=13.64d0,a11=0.0d0,a12=-50.0d0)
      double precision r3,l3
      parameter(r3=10.0d0,l3=0.2d0)
      double precision cos,atan
      intrinsic cos,atan
c
      x1dot=wdot(1)
      x2dot=wdot(2)
      x1=w(1)
      x2=w(2)
      x3=w(3)
      v1=w(4)
      v2=w(5)
      v3=w(6)
      z3=w(7)
      z4=w(8)
c
      delta(1)=x1+x2-x3
      delta(2)=r3*(x1+x2)+l3*(x1dot+x2dot)-v3
      delta(3)=100.0d0*cos(100.0d0*atan(1.0d0)*4.0d0*t)-v1
      delta(4)=-v1-v2
```

```
      delta(5)=v1-v3-z3
      delta(6)=v2-v3-z4

      if((x1.gt.0.0d0.or.v1.gt.v3).and.
     $   (x2.le.0.0d0.and.v2.lt.v3)) then
         delta(7)=(v1-a1*x1)/a2-x1dot
         delta(8)=0.0d0-x2dot
      else
         if((x2.gt.0.0d0.or.v2.gt.v3).and.
     $      (x1.lt.0.0d0.and.v1.lt.v3)) then
            delta(7)=0.0d0-x1dot
            delta(8)=(v2-a3*x2)/a4-x2dot
         else
            delta(7)=a5*v1+a6*v2+a7*x1+a8*x2-x1dot
            delta(8)=a9*v1+a10*v2+a11*x1+a12*x2-x2dot
         end if
      end if
c
      return
      end
```

A1 Analytical Derivative Code

A new model was created (see Section 2.2.2) and the file containing the source above was specified in the file list. The model was then translated and the following derivative code generated:

```
C###  - DAEPACK v1.0 - Copyright (C) M.I.T.
C###  - DERIVATIVE COMPUTATION -
      subroutine resad(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzderiv,zzzne,zzzirn,zzzjcn,zzziw)
!!independent { w, wdot }
!!dependent   { delta }
      implicit none
      double precision a8
      parameter(a8 = 0.0d0)
      double precision a9
      parameter(a9 = -11.64d0)
      integer neq
      integer ichvar
      double precision r3
      parameter(r3 = 10.0d0)
      double precision t
      double precision a12
      parameter(a12 = -50.0d0)
      integer ires
      double precision a10
      parameter(a10 = 13.64d0)
      double precision v1
      double precision a11
      parameter(a11 = 0.0d0)
      double precision v2
      double precision v3
      double precision x2dot
      double precision l3
      parameter(l3 = 0.2d0)
      double precision a2
```

```
      parameter(a2 = 0.24d0)
      double precision a3
      parameter(a3 = 12.0d0)
      double precision x1dot
      double precision z4
      double precision x1
      double precision a1
      parameter(a1 = 12.0d0)
      double precision x2
      double precision a6
      parameter(a6 = -11.64d0)
      double precision x3
      double precision a7
      parameter(a7 = -50.0d0)
      double precision a4
      parameter(a4 = 0.24d0)
      double precision a5
      parameter(a5 = 13.64d0)
      double precision z3
      double precision delta(neq)
      integer ipar(1)
      double precision w(neq)
      double precision rpar(1)
      double precision wdot(neq)
C###  Additional arguments for partial derivative computation
C###     zzzderiv - partial derivative array stored in sparse matrix
C###                format, pattern contained in zzzirn and zzzjcn.
C###     zzzne    - number of entries in zzzderiv.
C###     zzzirn   - row numbers for entries in zzzderiv.
C###     zzzjcn   - column numbers for entries in zzzderiv.
      double precision zzzderiv(*)
      integer zzzne,zzzirn(zzzne),zzzjcn(zzzne)
C###     zzziw    - integer workspace array.
      integer zzziw(*)
      integer zzzimodel
C###  Define elemenary variables and adjoints
      double precision zzzv1,zzzvbar1
      double precision zzzv2,zzzvbar2
      double precision zzzv3,zzzvbar3
      double precision zzzv4,zzzvbar4
      double precision zzzv5,zzzvbar5
      double precision zzzv6,zzzvbar6
      double precision zzzv7,zzzvbar7
      double precision zzzv8,zzzvbar8
      double precision zzzv9,zzzvbar9
      double precision zzzv10,zzzvbar10
      double precision zzzv11,zzzvbar11
      double precision zzzv12,zzzvbar12
      double precision zzzv13,zzzvbar13
      double precision zzzv14,zzzvbar14
      double precision zzzv15,zzzvbar15
      double precision zzzv16,zzzvbar16
      double precision zzzv17,zzzvbar17
C###
C###  Active variable offsets and loop control variables
      integer deltaoft
      integer woft
```

```
      integer wdotoft
      integer x1dotoft
      integer x2dotoft
      integer x1oft
      integer x2oft
      integer x3oft
      integer v1oft
      integer v2oft
      integer v3oft
      integer z3oft
      integer z4oft
      integer zzzn
      integer zzzm
      integer zzzi
      integer zzzindx
C###  Variable offsets for independent, dependent, and
C###  non-common block local active variables.
      deltaoft=0
      woft=deltaoft+neq
      wdotoft=woft+neq
      x1dotoft=wdotoft+neq
      x2dotoft=x1dotoft+1
      x1oft=x2dotoft+1
      x2oft=x1oft+1
      x3oft=x2oft+1
      v1oft=x3oft+1
      v2oft=v1oft+1
      v3oft=v2oft+1
      z3oft=v3oft+1
      z4oft=z3oft+1
C###  Number of independent and dependent variables.
      zzzn=neq+neq
      zzzm=neq
C###  Get unique identifier for this model
      call GETMDLID(zzzimodel,'resad')
C###  Create SVM for functions/subroutines.
      call CREATESV(zzzimodel,1,zzzn)
C###  Construct mapping between independent variable offsets and
C###  indices
      zzzindx=0
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=woft+zzzi
      end do
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=wdotoft+zzzi
      end do
C###  Initialize independent variable gradients to Cartesian basis
C###  vectors.
      call DSETIVCV(1,zzzn,zzziw)
C###
C###  Compute elementary variables
      zzzv1=wdot(1)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
```

```
      x1dot=zzzv1
C###
      call DSVM1(x1dotoft+1,1,
     $           zzzvbar1,wdotoft+1,1)
C###
C###  Compute elementary variables
      zzzv1=wdot(2)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      x2dot=zzzv1
C###
      call DSVM1(x2dotoft+1,1,
     $           zzzvbar1,wdotoft+2,1)
C###
C###  Compute elementary variables
      zzzv1=w(1)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      x1=zzzv1
C###
      call DSVM1(x1oft+1,1,
     $           zzzvbar1,woft+1,1)
C###
C###  Compute elementary variables
      zzzv1=w(2)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      x2=zzzv1
C###
      call DSVM1(x2oft+1,1,
     $           zzzvbar1,woft+2,1)
C###
C###  Compute elementary variables
      zzzv1=w(3)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      x3=zzzv1
C###
      call DSVM1(x3oft+1,1,
     $           zzzvbar1,woft+3,1)
C###
C###  Compute elementary variables
      zzzv1=w(4)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      v1=zzzv1
C###
      call DSVM1(v1oft+1,1,
     $           zzzvbar1,woft+4,1)
C###
C###  Compute elementary variables
      zzzv1=w(5)
```

```
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      v2=zzzv1
C###
      call DSVM1(v2oft+1,1,
     $           zzzvbar1,woft+5,1)
C###
C###  Compute elementary variables
      zzzv1=w(6)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      v3=zzzv1
C###
      call DSVM1(v3oft+1,1,
     $           zzzvbar1,woft+6,1)
C###
C###  Compute elementary variables
      zzzv1=w(7)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      z3=zzzv1
C###
      call DSVM1(z3oft+1,1,
     $           zzzvbar1,woft+7,1)
C###
C###  Compute elementary variables
      zzzv1=w(8)
C###  Compute elementary partial derivatives
      zzzvbar1=1.0d0
C###
      z4=zzzv1
C###
      call DSVM1(z4oft+1,1,
     $           zzzvbar1,woft+8,1)
C###
C###  Compute elementary variables
      zzzv1=x1
      zzzv2=x2
      zzzv3=zzzv1+zzzv2
      zzzv4=x3
      zzzv5=zzzv3-zzzv4
C###  Compute elementary partial derivatives
      zzzvbar5=1.0d0
      zzzvbar4=-zzzvbar5
      zzzvbar3=zzzvbar5
      zzzvbar2=zzzvbar3
      zzzvbar1=zzzvbar3
C###
      delta(1)=zzzv5
C###
      call DSVM3(deltaoft+1,1,
     $           zzzvbar4,x3oft+1,1,
     $           zzzvbar2,x2oft+1,1,
     $           zzzvbar1,x1oft+1,1)
```

```
C###
C###  Compute elementary variables
      zzzv1=r3
      zzzv2=x1
      zzzv3=x2
      zzzv4=zzzv2+zzzv3
      zzzv5=zzzv1*zzzv4
      zzzv6=l3
      zzzv7=x1dot
      zzzv8=x2dot
      zzzv9=zzzv7+zzzv8
      zzzv10=zzzv6*zzzv9
      zzzv11=zzzv5+zzzv10
      zzzv12=v3
      zzzv13=zzzv11-zzzv12
C###  Compute elementary partial derivatives
      zzzvbar13=1.0d0
      zzzvbar12=-zzzvbar13
      zzzvbar11=zzzvbar13
      zzzvbar10=zzzvbar11
      zzzvbar9=zzzvbar10*zzzv6
      zzzvbar8=zzzvbar9
      zzzvbar7=zzzvbar9
      zzzvbar5=zzzvbar11
      zzzvbar4=zzzvbar5*zzzv1
      zzzvbar3=zzzvbar4
      zzzvbar2=zzzvbar4
C###
      delta(2)=zzzv13
C###
      call DSVM5(deltaoft+2,1,
     $           zzzvbar12,v3oft+1,1,
     $           zzzvbar8,x2dotoft+1,1,
     $           zzzvbar7,x1dotoft+1,1,
     $           zzzvbar3,x2oft+1,1,
     $           zzzvbar2,x1oft+1,1)
C###
C###  Compute elementary variables
      zzzv1=100.0d0
      zzzv2=100.0d0
      zzzv3=1.0d0
      zzzv4=atan(zzzv3)
      zzzv5=zzzv2*zzzv4
      zzzv6=4.0d0
      zzzv7=zzzv5*zzzv6
      zzzv8=t
      zzzv9=zzzv7*zzzv8
      zzzv10=cos(zzzv9)
      zzzv11=zzzv1*zzzv10
      zzzv12=v1
      zzzv13=zzzv11-zzzv12
C###  Compute elementary partial derivatives
      zzzvbar13=1.0d0
      zzzvbar12=-zzzvbar13
C###
      delta(3)=zzzv13
C###
```

```
      call DSVM1(deltaoft+3,1,
     $           zzzvbar12,v1oft+1,1)
C###
C###  Compute elementary variables
      zzzv1=v1
      zzzv2=-zzzv1
      zzzv3=v2
      zzzv4=zzzv2-zzzv3
C###  Compute elementary partial derivatives
      zzzvbar4=1.0d0
      zzzvbar3=-zzzvbar4
      zzzvbar2=zzzvbar4
      zzzvbar1=-zzzvbar2
C###
      delta(4)=zzzv4
C###
      call DSVM2(deltaoft+4,1,
     $           zzzvbar3,v2oft+1,1,
     $           zzzvbar1,v1oft+1,1)
C###
C###  Compute elementary variables
      zzzv1=v1
      zzzv2=v3
      zzzv3=zzzv1-zzzv2
      zzzv4=z3
      zzzv5=zzzv3-zzzv4
C###  Compute elementary partial derivatives
      zzzvbar5=1.0d0
      zzzvbar4=-zzzvbar5
      zzzvbar3=zzzvbar5
      zzzvbar2=-zzzvbar3
      zzzvbar1=zzzvbar3
C###
      delta(5)=zzzv5
C###
      call DSVM3(deltaoft+5,1,
     $           zzzvbar4,z3oft+1,1,
     $           zzzvbar2,v3oft+1,1,
     $           zzzvbar1,v1oft+1,1)
C###
C###  Compute elementary variables
      zzzv1=v2
      zzzv2=v3
      zzzv3=zzzv1-zzzv2
      zzzv4=z4
      zzzv5=zzzv3-zzzv4
C###  Compute elementary partial derivatives
      zzzvbar5=1.0d0
      zzzvbar4=-zzzvbar5
      zzzvbar3=zzzvbar5
      zzzvbar2=-zzzvbar3
      zzzvbar1=zzzvbar3
C###
      delta(6)=zzzv5
C###
      call DSVM3(deltaoft+6,1,
     $           zzzvbar4,z4oft+1,1,
```

```
     $           zzzvbar2,v3oft+1,1,
     $           zzzvbar1,v2oft+1,1)
C###
      if((x1.gt.0.0d0.or.v1.gt.v3).and.(x2.le.0.0d0.and.v2.lt.v3))then
C###  Compute elementary variables
         zzzv1=v1
         zzzv2=a1
         zzzv3=x1
         zzzv4=zzzv2*zzzv3
         zzzv5=zzzv1-zzzv4
         zzzv6=a2
         zzzv7=zzzv5/zzzv6
         zzzv8=x1dot
         zzzv9=zzzv7-zzzv8
C###  Compute elementary partial derivatives
         zzzvbar9=1.0d0
         zzzvbar8=-zzzvbar9
         zzzvbar7=zzzvbar9
         zzzvbar5=zzzvbar7/zzzv6
         zzzvbar4=-zzzvbar5
         zzzvbar3=zzzvbar4*zzzv2
         zzzvbar1=zzzvbar5
C###
         delta(7)=zzzv9
C###
         call DSVM3(deltaoft+7,1,
     $              zzzvbar8,x1dotoft+1,1,
     $              zzzvbar3,x1oft+1,1,
     $              zzzvbar1,v1oft+1,1)
C###
C###  Compute elementary variables
         zzzv1=0.0d0
         zzzv2=x2dot
         zzzv3=zzzv1-zzzv2
C###  Compute elementary partial derivatives
         zzzvbar3=1.0d0
         zzzvbar2=-zzzvbar3
C###
         delta(8)=zzzv3
C###
         call DSVM1(deltaoft+8,1,
     $              zzzvbar2,x2dotoft+1,1)
C###
      else
         if((x2.gt.0.0d0.or.v2.gt.v3).and.(x1.lt.0.0d0.and.v1.lt.
     $      v3))then
C###  Compute elementary variables
            zzzv1=0.0d0
            zzzv2=x1dot
            zzzv3=zzzv1-zzzv2
C###  Compute elementary partial derivatives
            zzzvbar3=1.0d0
            zzzvbar2=-zzzvbar3
C###
            delta(7)=zzzv3
C###
            call DSVM1(deltaoft+7,1,
```

```
     $                 zzzvbar2,x1dotoft+1,1)
C###
C###  Compute elementary variables
            zzzv1=v2
            zzzv2=a3
            zzzv3=x2
            zzzv4=zzzv2*zzzv3
            zzzv5=zzzv1-zzzv4
            zzzv6=a4
            zzzv7=zzzv5/zzzv6
            zzzv8=x2dot
            zzzv9=zzzv7-zzzv8
C###  Compute elementary partial derivatives
            zzzvbar9=1.0d0
            zzzvbar8=-zzzvbar9
            zzzvbar7=zzzvbar9
            zzzvbar5=zzzvbar7/zzzv6
            zzzvbar4=-zzzvbar5
            zzzvbar3=zzzvbar4*zzzv2
            zzzvbar1=zzzvbar5
C###
            delta(8)=zzzv9
C###
            call DSVM3(deltaoft+8,1,
     $                 zzzvbar8,x2dotoft+1,1,
     $                 zzzvbar3,x2oft+1,1,
     $                 zzzvbar1,v2oft+1,1)
C###
         else
C###  Compute elementary variables
            zzzv1=a5
            zzzv2=v1
            zzzv3=zzzv1*zzzv2
            zzzv4=a6
            zzzv5=v2
            zzzv6=zzzv4*zzzv5
            zzzv7=zzzv3+zzzv6
            zzzv8=a7
            zzzv9=x1
            zzzv10=zzzv8*zzzv9
            zzzv11=zzzv7+zzzv10
            zzzv12=a8
            zzzv13=x2
            zzzv14=zzzv12*zzzv13
            zzzv15=zzzv11+zzzv14
            zzzv16=x1dot
            zzzv17=zzzv15-zzzv16
C###  Compute elementary partial derivatives
            zzzvbar17=1.0d0
            zzzvbar16=-zzzvbar17
            zzzvbar15=zzzvbar17
            zzzvbar14=zzzvbar15
            zzzvbar13=zzzvbar14*zzzv12
            zzzvbar11=zzzvbar15
            zzzvbar10=zzzvbar11
            zzzvbar9=zzzvbar10*zzzv8
            zzzvbar7=zzzvbar11
```

```
            zzzvbar6=zzzvbar7
            zzzvbar5=zzzvbar6*zzzv4
            zzzvbar3=zzzvbar7
            zzzvbar2=zzzvbar3*zzzv1
C###
            delta(7)=zzzv17
C###
            call DSVM5(deltaoft+7,1,
     $                  zzzvbar16,x1dotoft+1,1,
     $                  zzzvbar13,x2oft+1,1,
     $                  zzzvbar9,x1oft+1,1,
     $                  zzzvbar5,v2oft+1,1,
     $                  zzzvbar2,v1oft+1,1)
C###
C###  Compute elementary variables
            zzzv1=a9
            zzzv2=v1
            zzzv3=zzzv1*zzzv2
            zzzv4=a10
            zzzv5=v2
            zzzv6=zzzv4*zzzv5
            zzzv7=zzzv3+zzzv6
            zzzv8=a11
            zzzv9=x1
            zzzv10=zzzv8*zzzv9
            zzzv11=zzzv7+zzzv10
            zzzv12=a12
            zzzv13=x2
            zzzv14=zzzv12*zzzv13
            zzzv15=zzzv11+zzzv14
            zzzv16=x2dot
            zzzv17=zzzv15-zzzv16
C###  Compute elementary partial derivatives
            zzzvbar17=1.0d0
            zzzvbar16=-zzzvbar17
            zzzvbar15=zzzvbar17
            zzzvbar14=zzzvbar15
            zzzvbar13=zzzvbar14*zzzv12
            zzzvbar11=zzzvbar15
            zzzvbar10=zzzvbar11
            zzzvbar9=zzzvbar10*zzzv8
            zzzvbar7=zzzvbar11
            zzzvbar6=zzzvbar7
            zzzvbar5=zzzvbar6*zzzv4
            zzzvbar3=zzzvbar7
            zzzvbar2=zzzvbar3*zzzv1
C###
            delta(8)=zzzv17
C###
            call DSVM5(deltaoft+8,1,
     $                  zzzvbar16,x2dotoft+1,1,
     $                  zzzvbar13,x2oft+1,1,
     $                  zzzvbar9,x1oft+1,1,
     $                  zzzvbar5,v2oft+1,1,
     $                  zzzvbar2,v1oft+1,1)
C###
         end if
```

```
      end if
      goto 11111
C###  Construct derivative matrix and sparsity pattern before
C###  returning.
11111 continue
C###  Construct mapping between dependent variable offsets and indices
      zzzindx=0
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=deltaoft+zzzi
      end do
      call DCPRVS(1,zzzm,zzziw,zzzderiv,zzzne,zzzirn,zzzjcn)
C###  Delete SVM for functions/subroutines.
      call DELETESV(zzzimodel,1)
C###  Done
      return
      end
```

The derivative code generation options specified were **Jacobian alone** and **Sparse matrix**. The suffix used for generated code was set to "ad". With these options, the new root program unit name is `resad` and has the following interface,

```
      subroutine resad(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzderiv,zzzne,zzzirn,zzzjcn,zzziw).
```

There are five additional arguments: `zzzderiv`, `zzzne`, `zzzirn`, `zzzjcn`, and `zzziw`. Real array `zzzderiv` contains the entries in the Jacobian matrix (in this case, partial derivatives of `delta` with respect to `w` and `wdot`) that are not identically zero (based on structural information), `zzzne` returns the number of nonzero entries, `zzzirn` and `zzzjcn` contain the row and column number, respectively, corresponding to the derivative entries in `zzzderiv`, and `zzziw` is integer workspace of length equal to max{number of independent variables, number of dependent variables}. The sparsity pattern is sorted by rows (i.e., all of the entries in row 1 followed by all of the entries in row 2 and so on) and the column numbers are sorted by increasing number within the rows. The user must specify none of these additional arguments prior to calling `resad`. The arrays `zzzderiv`, `zzzirn`, and `zzzjcn` must be dimensioned larger than or equal to the largest value of `zzzne` expected. Since this problem is small, little is gained from using the sparse matrix format. By specifying the options **Jacobian alone** and **Dense matrix**, the following interface for the root program unit is generated:

```
      subroutine resad(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzldm,zzzderiv,zzziw).
```

Only three additional arguments are appended to the original argument list: `zzzldm`, `zzzderiv`, and `zzziw`. Integer argument `zzzldm` is the leading dimension of the two dimensional real array `zzzderiv`, `zzzderiv` contains the Jacobian matrix as a dense two dimensional array, and `zzziw` is the integer workspace (same dimension as described above). If Jacobian-matrix products are desired, then the following interfaces are created when **Jacobian-matrix product** and **Sparse matrix** and **Jacobian-matrix product** and **Dense matrix** are specified as options:

```
      subroutine resad(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzderiv,zzzne,zzzirn,zzzjcn,zzznscol,zzzseed,zzziw)
```

and

```
      subroutine resad(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzldm,zzzderiv,zzznscol,zzzseed,zzziw).
```

The argument lists of the two interfaces above are identical to those described above with two additional arguments: `zzznscol` and `zzzseed`. The values returned in `zzzderiv` are

$$J(x)S$$

where S has `N` rows and `zzznscol` columns (`N` is the number of independent variables). The transpose of S is stored in `zzzseed` (i.e., `zzzseed` is dimensioned as `zzzseed(zzznscol,N)` and its rows are filled with the columns of S). The transpose of the matrix S is required for efficiency considerations (the gradients of the independent variables are initialized with the contiguous columns of `zzzseed`).

Inspecting the generated code, one finds several subroutines present that were not in the original code. These are utility routines used by DAEPACK to perform various operations (e.g., creating active variable tables, performing sparse SAXPY operations, etc.). These additional routines are provided in a library that must be linked into any application using the generated code. Several versions of the library are available on a variety of platforms. The generated code is as platform independent as the original source code. The code can be generated on a PC running Windows then ported to a Unix box where it can be compiled and linked (along with the appropriate utility library) into a larger application. Calling the modified root program unit will produce the information obtained from the original root program unit in addition to the derivative information. In the sparse case, the sparsity pattern corresponds to the mode (see Section 2.4.3) determined by the current input. In the sparse Jacobian-matrix product case, the returned sparsity pattern corresponds to the Jacobian-matrix product.

A2 Sparsity Pattern Code

Although the sparse Jacobian code returns the sparsity pattern, there may be situations when only the sparsity pattern is desired (e.g., determining degrees of freedom during model development, structural index analysis, and other structural algorithms). In this case, code can be generated that returns only the sparsity pattern of the Jacobian matrix. The following sparsity pattern code was generated from the original model (where `w` and `wdot` are the independent variables, `delta` are the dependent variables, and the suffix for generated code was set to "sp"):

```
C###  DAEPACK v0.02 - Copyright (C) M.I.T.
C###  - SPARSITY PATTERN GENERATOR -
      subroutine ressp(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $ zzzne,zzzirn,zzzjcn,zzziw)
!!independent { w, wdot }
!!dependent   { delta }
C###
      implicit none
      double precision a8
      parameter(a8 = 0.0d0)
      double precision a9
      parameter(a9 = -11.64d0)
      integer neq
      integer ichvar
      double precision r3
      parameter(r3 = 10.0d0)
      double precision t
      double precision a12
      parameter(a12 = -50.0d0)
      integer ires
      double precision a10
      parameter(a10 = 13.64d0)
      double precision v1
      double precision a11
      parameter(a11 = 0.0d0)
      double precision v2
      double precision v3
      double precision x2dot
      double precision l3
      parameter(l3 = 0.2d0)
      double precision a2
      parameter(a2 = 0.24d0)
      double precision a3
      parameter(a3 = 12.0d0)
      double precision x1dot
```

```
      double precision z4
      double precision x1
      double precision a1
      parameter(a1 = 12.0d0)
      double precision x2
      double precision a6
      parameter(a6 = -11.64d0)
      double precision x3
      double precision a7
      parameter(a7 = -50.0d0)
      double precision a4
      parameter(a4 = 0.24d0)
      double precision a5
      parameter(a5 = 13.64d0)
      double precision z3
      double precision delta(neq)
      integer ipar(1)
      double precision w(neq)
      double precision rpar(1)
      double precision wdot(neq)
C###  Additional arguments for occurrence information construction
C###     zzzne  - number of entries in incidence matrix
C###     zzzirn - row numbers
C###     zzzjcn - column numbers
C###     zzziw  - integer workspace
      integer zzzne,zzzirn(zzzne),zzzjcn(zzzne),zzziw(*)
      external CREATEOI,SETIVOI,RSTOIW,MERGEOI,ADJOIW,COMPROI
C###  Active variable offsets and loop control variables
      integer deltaoft
      integer woft
      integer wdotoft
      integer x1dotoft
      integer x2dotoft
      integer x1oft
      integer x2oft
      integer x3oft
      integer v1oft
      integer v2oft
      integer v3oft
      integer z3oft
      integer z4oft
      integer zzzn
      integer zzzm
      integer zzzi
      integer zzzindx
C###  Variable offsets for independent, dependent, and
C###  non-common block local active variables.
      deltaoft=0
      woft=deltaoft+neq
      wdotoft=woft+neq
      x1dotoft=wdotoft+neq
      x2dotoft=x1dotoft+1
      x1oft=x2dotoft+1
      x2oft=x1oft+1
      x3oft=x2oft+1
      v1oft=x3oft+1
      v2oft=v1oft+1
```

```
      v3oft=v2oft+1
      z3oft=v3oft+1
      z4oft=z3oft+1
C###  Number of independent and dependent variables.
      zzzn=neq+neq
      zzzm=neq
C###  Create OIM for functions/subroutines.
      call CREATEOI(1,zzzn)
C### Construct mapping between independent variable offsets and indices
      zzzindx=0
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=woft+zzzi
      end do
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=wdotoft+zzzi
      end do
C###  Set independent variable numbers
      call SETIVOI(1,zzzn,zzziw)
      x1dot=wdot(1)
      call RSTOIW()
      call MERGEOI(wdotoft+1,1)
      call ADJOIW(x1dotoft+1,1)
      x2dot=wdot(2)
      call RSTOIW()
      call MERGEOI(wdotoft+2,1)
      call ADJOIW(x2dotoft+1,1)
      x1=w(1)
      call RSTOIW()
      call MERGEOI(woft+1,1)
      call ADJOIW(x1oft+1,1)
      x2=w(2)
      call RSTOIW()
      call MERGEOI(woft+2,1)
      call ADJOIW(x2oft+1,1)
      x3=w(3)
      call RSTOIW()
      call MERGEOI(woft+3,1)
      call ADJOIW(x3oft+1,1)
      v1=w(4)
      call RSTOIW()
      call MERGEOI(woft+4,1)
      call ADJOIW(v1oft+1,1)
      v2=w(5)
      call RSTOIW()
      call MERGEOI(woft+5,1)
      call ADJOIW(v2oft+1,1)
      v3=w(6)
      call RSTOIW()
      call MERGEOI(woft+6,1)
      call ADJOIW(v3oft+1,1)
      z3=w(7)
      call RSTOIW()
      call MERGEOI(woft+7,1)
      call ADJOIW(z3oft+1,1)
      z4=w(8)
```

```
 call RSTOIW()
 call MERGEOI(woft+8,1)
 call ADJOIW(z4oft+1,1)
 delta(1)=x1+x2-x3
 call RSTOIW()
 call MERGEOI(x1oft+1,1)
 call MERGEOI(x2oft+1,1)
 call MERGEOI(x3oft+1,1)
 call ADJOIW(deltaoft+1,1)
 delta(2)=r3*(x1+x2)+l3*(x1dot+x2dot)-v3
 call RSTOIW()
 call MERGEOI(x1oft+1,1)
 call MERGEOI(x2oft+1,1)
 call MERGEOI(x1dotoft+1,1)
 call MERGEOI(x2dotoft+1,1)
 call MERGEOI(v3oft+1,1)
 call ADJOIW(deltaoft+2,1)
 delta(3)=100.0d0*cos(100.0d0*atan(1.0d0)*4.0d0*t)-v1
 call RSTOIW()
 call MERGEOI(v1oft+1,1)
 call ADJOIW(deltaoft+3,1)
 delta(4)=-v1-v2
 call RSTOIW()
 call MERGEOI(v1oft+1,1)
 call MERGEOI(v2oft+1,1)
 call ADJOIW(deltaoft+4,1)
 delta(5)=v1-v3-z3
 call RSTOIW()
 call MERGEOI(v1oft+1,1)
 call MERGEOI(v3oft+1,1)
 call MERGEOI(z3oft+1,1)
 call ADJOIW(deltaoft+5,1)
 delta(6)=v2-v3-z4
 call RSTOIW()
 call MERGEOI(v2oft+1,1)
 call MERGEOI(v3oft+1,1)
 call MERGEOI(z4oft+1,1)
 call ADJOIW(deltaoft+6,1)
 if((x1.gt.0.0d0.or.v1.gt.v3).and.(x2.le.0.0d0.and.v2.lt.v3))
$   then
    delta(7)=(v1-a1*x1)/a2-x1dot
    call RSTOIW()
    call MERGEOI(v1oft+1,1)
    call MERGEOI(x1oft+1,1)
    call MERGEOI(x1dotoft+1,1)
    call ADJOIW(deltaoft+7,1)
    delta(8)=0.0d0-x2dot
    call RSTOIW()
    call MERGEOI(x2dotoft+1,1)
    call ADJOIW(deltaoft+8,1)
 else
    if((x2.gt.0.0d0.or.v2.gt.v3).and.(x1.lt.0.0d0.and.v1.lt.
$      v3))then
       delta(7)=0.0d0-x1dot
       call RSTOIW()
       call MERGEOI(x1dotoft+1,1)
       call ADJOIW(deltaoft+7,1)
```

```
            delta(8)=(v2-a3*x2)/a4-x2dot
            call RSTOIW()
            call MERGEOI(v2oft+1,1)
            call MERGEOI(x2oft+1,1)
            call MERGEOI(x2dotoft+1,1)
            call ADJOIW(deltaoft+8,1)
         else
            delta(7)=a5*v1+a6*v2+a7*x1+a8*x2-x1dot
            call RSTOIW()
            call MERGEOI(v1oft+1,1)
            call MERGEOI(v2oft+1,1)
            call MERGEOI(x1oft+1,1)
            call MERGEOI(x2oft+1,1)
            call MERGEOI(x1dotoft+1,1)
            call ADJOIW(deltaoft+7,1)
            delta(8)=a9*v1+a10*v2+a11*x1+a12*x2-x2dot
            call RSTOIW()
            call MERGEOI(v1oft+1,1)
            call MERGEOI(v2oft+1,1)
            call MERGEOI(x1oft+1,1)
            call MERGEOI(x2oft+1,1)
            call MERGEOI(x2dotoft+1,1)
            call ADJOIW(deltaoft+8,1)
         end if
      end if
      goto 11111
C###  Compress occurrence information before returning
11111 continue
C###  Construct mapping between dependent variable offsets and indices
      zzzindx=0
      do zzzi=1,neq
         zzzindx=zzzindx+1
         zzziw(zzzindx)=deltaoft+zzzi
      end do
      call COMPROI(1,zzzm,zzziw,zzzne,zzzirn,zzzjcn)
C###  Delete OIM for functions/subroutines.
      call DELETEOI(1)
C###  Done
      return
      end
```

The new argument list contains four additional arguments: `zzzne`, `zzzirn`, `zzzjcn`, and `zzziw`. As in the sparse Jacobian case, the number of entries in the sparsity pattern is returned in `zzzne` and `zzzirn` and `zzzjcn` contains the row and column numbers, respectively, of the sparsity pattern. The sparsity pattern is sorted by rows (i.e., all of the entries in row 1 followed by all of the entries in row 2 and so on) and the column numbers are sorted by increasing number within the rows. The integer workspace array `zzziw` must be dimensioned to a value greater than or equal to the number of independent or dependent variables, whichever is larger. Integer arrays and `zzzirn` and `zzzjcn` must be dimensioned to at least the largest value expected for `zzzne`, however, none of the variables must be set by the user prior to calling the generated code. As with the derivative code described above, the sparsity pattern code is as platform independent as the original code. Upon compiling, the sparsity pattern code must be linked into the larger application with the appropriate DAEPACK utility library.

A3 Discontinuity-Locked Models

Once generated, the modeler can use the derivative and sparsity pattern codes very easily. All he or she must do is make sure that sufficient memory has been allocated for the additional arrays in the argument

list. If a Jacobian-matrix product is desired then `zzznscol` must be specified and the rows of the argument `zzzseed` must be filled with the columns of the matrix *S*. The situation is a bit more complicated for the discontinuity locking code. The code generated for discontinuity locking is shown below.

```
C###  - DAEPACK v1.0 - Copyright (C) M.I.T.
C###  - DISCONTINUITY LOCKED MODEL -
      subroutine resdl(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $   zzznsc,zzzndf,zzzisc,zzziscchng,zzzdiscon,zzzlocked,
     $   zzzistate,zzzig)
C###
      implicit none
      double precision a8
      parameter(a8 = 0.0d0)
      double precision a9
      parameter(a9 = -11.64d0)
      integer neq
      integer ichvar
      double precision r3
      parameter(r3 = 10.0d0)
      double precision t
      double precision a12
      parameter(a12 = -50.0d0)
      integer ires
      double precision a10
      parameter(a10 = 13.64d0)
      double precision v1
      double precision a11
      parameter(a11 = 0.0d0)
      double precision v2
      double precision v3
      double precision x2dot
      double precision l3
      parameter(l3 = 0.2d0)
      double precision a2
      parameter(a2 = 0.24d0)
      double precision a3
      parameter(a3 = 12.0d0)
      double precision x1dot
      double precision z4
      double precision x1
      double precision a1
      parameter(a1 = 12.0d0)
      double precision x2
      double precision a6
      parameter(a6 = -11.64d0)
      double precision x3
      double precision a7
      parameter(a7 = -50.0d0)
      double precision a4
      parameter(a4 = 0.24d0)
      double precision a5
      parameter(a5 = 13.64d0)
      double precision z3
      double precision delta(neq)
      integer ipar(1)
```

```
      double precision w(neq)
      double precision rpar(1)
      double precision wdot(neq)
C###
C###  Additional argument for discontinuity locked evaluations
C###           zzznsc - Total number of currently active state
C###                    conditions.
C###           zzzndf - Total number of currently active discontinuity
C###                    functions.
C###    zzzisc(zzznsc) - Number of discontinuity functions in each state
C###                    condition.
C### zzziscchng(zzznsc) - Contains indices of state conditions that
C###                    have flipped.
C### zzzdiscon(zzzndf) - Current values of discontinuity functions.
C###        zzzlocked - Flag indicating whether or not this is a locked
C###                    evaluation.
C###        zzzistate - State index to determine whether or not it has
C###                    flipped.
C###          ig(ndf) - Values of discontinuity functions to determine
C###                    whether or not state condition has flipped.
      integer zzznsc,zzzndf,zzzisc(zzznsc)
      intger zzziscchng(zzznsc),zzzistate,zzzig(zzzndf),izzzchng
      double precision zzzdiscon(zzzndf)
      logical logval,locked,lzzzdlk
      external lzzzdlk,izzzchng
C###
      zzznsc=0
      zzzndf=0
C###
      x1dot=wdot(1)
      x2dot=wdot(2)
      x1=w(1)
      x2=w(2)
      x3=w(3)
      v1=w(4)
      v2=w(5)
      v3=w(6)
      z3=w(7)
      z4=w(8)
      delta(1)=x1+x2-x3
      delta(2)=r3*(x1+x2)+l3*(x1dot+x2dot)-v3
      delta(3)=100.0d0*cos(100.0d0*atan(1.0d0)*4.0d0*t)-v1
      delta(4)=-v1-v2
      delta(5)=v1-v3-z3
      delta(6)=v2-v3-z4
C###  Modified if equation for discontinuity locking
      zzznsc=zzznsc+1
      if(zzznsc.eq. zzzistate) then
         logval=(ig(1).eq.1.or.ig(2).eq.1).and.(ig(3).eq.1.and.
     $      ig(4).eq.1)
         zzzistate=izzzchng(zzzistate,logval)
         return
      else
         zzzndf=zzzndf+1
         zzzdiscon(zzzndf)=x1-(0.0d0)
         zzzndf=zzzndf+1
         zzzdiscon(zzzndf)=v1-(v3)
```

```
         zzzndf=zzzndf+1
         zzzdiscon(zzzndf)=0.0d0-(x2)
         zzzndf=zzzndf+1
         zzzdiscon(zzzndf)=v3-(v2)
         if(.not. zzzlocked) zzzisc(zzznsc)=4
       end if
       logval=(x1.gt.0.0d0.or.v1.gt.v3).and.(x2.le.0.0d0.and.v2.lt.
     $   v3)
       if(lzzzdlk(zzzlocked, zzznsc, zzzlogval, zzziscchng)) then
         delta(7)=(v1-a1*x1)/a2-x1dot
         delta(8)=0.0d0-x2dot
       else
C###   Modified if equation for discontinuity locking
         zzznsc=zzznsc+1
         if(zzznsc.eq. zzzistate) then
           logval=(ig(1).eq.1.or.ig(2).eq.1).and.(ig(3).eq.1.and.
     $        ig(4).eq.1)
           zzzistate=izzzchng(zzzistate,logval)
           return
         else
           zzzndf=zzzndf+1
           zzzdiscon(zzzndf)=x2-(0.0d0)
           zzzndf=zzzndf+1
           zzzdiscon(zzzndf)=v2-(v3)
           zzzndf=zzzndf+1
           zzzdiscon(zzzndf)=0.0d0-(x1)
           zzzndf=zzzndf+1
           zzzdiscon(zzzndf)=v3-(v1)
           if(.not. zzzlocked) zzzisc(zzznsc)=4
         end if
         logval=(x2.gt.0.0d0.or.v2.gt.v3).and.(x1.lt.0.0d0.and.v1.lt.
     $      v3)
         if(lzzzdlk(zzzlocked, zzznsc, zzzlogval, zzziscchng)) then
           delta(7)=0.0d0-x1dot
           delta(8)=(v2-a3*x2)/a4-x2dot
         else
           delta(7)=a5*v1+a6*v2+a7*x1+a8*x2-x1dot
           delta(8)=a9*v1+a10*v2+a11*x1+a12*x2-x2dot
         end if
       end if
       return
C###   Done
       end
```

Unlike the derivative and sparsity pattern codes, the user need not specify independent and dependent variables. The main options that must be specified by the user are the root program unit, the suffix used for generated code ("dl" in the example above), the name of the generated file, and the target language (currently the only option available is Fortran).

The interface of the new main program unit has an additional eight variables in the argument list:

```
       subroutine resdl(neq,t,w,wdot,delta,ires,ichvar,rpar,ipar,
     $   zzznsc,zzzndf,zzzisc,zzziscchng,zzzdiscon,zzzlocked,
     $   zzzistate,zzzig).
```

The example model shown above contains at most two state conditions each containing four discontinuity functions (see the IF equations in the original model). (If the first state condition is locked at true then the second state condition will be hidden.) Integer arguments `zzznsc` and `zzzndf` return the total number of

active state conditions and discontinuity functions, respectively, and need not be set by the user. Integer array argument `zzzisc(zzznsc)` contains the number of discontinuity functions in each state condition. It does not have to be initialized by the user but must be dimensioned to a value greater than or equal to the maximum number of state conditions expected. Integer array argument `zzziscchng(zzznsc)` contains the indices (if any) of state conditions that have changed state from the last time the model was called. This array is important because not all state conditions can be decomposed into a set of real-valued discontinuity functions. For example, the state conditions may contain discrete (integer) conditions that may be changed by the user between successive calls to the model. The real array `zzzdiscon(zzzndf)` contains the values of the discontinuity functions where the associated relational expression have been converted to the form:

$$g_{ij} \geq (>)\ 0$$

where g_{ij} is the *j*-th discontinuity function of the *i*-th state condition. Discontinuity functions are created from real-valued relational expression of the form:

Real Expression $\{\geq, >, \leq, <\}$ Real Expression

If a state condition contains logical atoms different than that above then the switching of a state condition is identified by the values returned in `zzziscchng`. The user must specify the logical variable `zzzlocked` before calling the discontinuity-locked model. If the model is called with `zzzlocked` equal to false then the model is evaluated as if the discontinuous equations were not locked and the current mode (i.e., the logical values of all of the state conditions) is recorded (with very little additional overhead). When the discontinuity-locked model is called with `zzzlocked` equal to true then mode active the last time the model was called with `zzzlocked` equal to false is returned. For example, suppose the original model contain the following segment of code:

```
IF(X > 10.0) THEN
   Y = LOG(X)/Z
ELSE
   Y = 0.0
END IF
```

Further, suppose the model is called with `zzzlocked` equal to false and `X` equal to `20.0` then `Y` is set equal to `LOG(X)/Z`. If the model is then called with `zzzlocked` equal to true then `Y` will be set equal to `LOG(X)/Z` regardless of the actual value of `X`. Care must be taken if IF equations are used to avoid evaluating expression at arguments outside their domain (e.g., if the code above is executed with `X` less than or equal to zero). In this case, the argument `zzziscchng` can be used to determine if this situation has occurred. The remaining two arguments, `zzzistate` and `zzzig(zzzndf)`, can be used within a state event location algorithm. For example, the algorithm employed in the DAEPACK component DSL48E identifies state events by augmenting the original DAE with equations of the form:

$$z_{ij} = g_{ij},$$

corresponding to the currently active discontinuity functions. State events are identified between mesh points by searching for roots of the discontinuity function over the previous time step using interpolating polynomial (such as those available when using the BDF method). If a root is identified then the algorithm must know whether or not this corresponds to a state event. The discontinuity-locked model is called with `zzzlocked` equal to true and `zzzistate` equal to the state condition of interest (i.e., the state condition containing the discontinuity function that has crossed zero). The array `zzzig` will have entries corresponding to the discontinuity functions in this state condition set to 1 if the discontinuity function is true (i.e., greater than zero) or 0 otherwise (i.e., less than zero). If the return value of `zzzistate` is 1 then the state event has occurred at the interpolated point, otherwise `zzzistate` is set to 0. An in-depth discussion of the use of this discontinuity-locked model is presented in the DAEPACK DSL48E manual.

Appendix B. Illustrative Example

Appendix C. Command-line Interface

As mentioned in the text, the DAEPACK code generation components are accessed through a command-line interface (CLI) on platforms other than Windows 9x and Windows NT. Although future releases of DAEPACK will provide GUIs on these platforms, the CLI provides a convenient way to rapidly generate code without having to maneuver through several windows or forms. The manual provides a description of the various code generation options that are mentioned below.

As implied by the name, the CLI is evoked from the command-line. The Backus-normal-like description of the command-line used to generate code is

```
% daepack –s[pec] spec-file (-m model-file | [Fortran source file]+)
```

The command `daepack` starts the DAEPACK translator and code generator. The options `–s[pec] spec-file` defines the specification file (described below). The notation `–s[pec]` means either `–s` or `–spec` is acceptable notation. The notation `(-m model-file | [Fortran source file]+)` means that either you may specify a model file (`-m model-file`) *or* a list of one or more Fortran source files (`[Fortran source file]+`). As described in the manual, a model file is simply a text file of the form:

```
N
Fortran-source-file-name-1
Fortran-source-file-name-2
.
.
.
Fortran-source-file-name-N
```

Whichever form is chosen, the model file or the list of Fortran source files, the names specified must include the path (relative to the location where `daepack` is executed).

The options required when generating code are specified through the specification file. A specification file is a text file containing one or more of the following specification sections:

```
GENERATE (DERIVS | SPARSITY | DISCONLOCK)
# This is a comment that can appear anywhere.
ROOT: root-program-unit-name  # This is another comment.
SUFFIX: suffix
OUTFILE: generated-file-name
[INDEPENDENT: comma-separated-variable-list]
[DEPENDENT: comma-separated-variable-list]
[JACOBIANXMATRIX: (TRUE | FALSE)]
[SPARSESTORAGE: (TRUE | FALSE)]
END
```

The first line of the specification section indicates which type of code is to be generated. The last line closes a specification section. All options within the specification section may appear in any order. The options contained in brackets are optional and are ignored if not applicable (e.g., `JACOBIANXMATRIX` option is not used when generating sparsity patterns or discontinuity-locked models). Comments begin with the '#' symbol and continue for the remainder of the line. They may appear anywhere within the specification file. The options available should seem familiar after reading the manual. The `ROOT` option allows the user to specify the name of the root program unit (i.e., entry point of model evaluation). The name specified must be a program unit name (subroutine or function) with global scope contained in the source files specified. The `SUFFIX` option allows the user to specify the suffix that will be appended to the name of all generated program units (e.g., `res` becomes `resad` when suffix is "ad"). The `OUTFILE`

option allows the user to specify the name of the generated file. If derivative or sparsity pattern code is to be generated then the INDEPENDENT and DEPENDENT options can be used to specify lists of independent and dependent variables. These options will override any independent or dependent variable specifications already contained in the code (using DAEPACK preprocessor directives). The variable lists must contain the valid Fortran names of arguments, local variables and/or common block variables contained in the root program unit. The variables specified may be scalars, array elements, array slices, or full arrays. The JACOBIANMATRIX option is set to TRUE if code is to be generated that returns the product of the Jacobian and an arbitrary conformable matrix. Setting this option to FALSE generates code that returns the Jacobian matrix alone. The SPARSESTORAGE option is set to TRUE if the derivative matrix is to be returned in sparse triplet form (i.e., coordinate form) or set to FALSE if a dense two-dimensional matrix is desired.

The specification file must contain one or more specification sections and code is generated for each section present. Care must be taken when naming the output files in each section; a file will be overwritten if the name appears more then once as an OUTFILE option.

Appendix D. Specifying Active Array Dimensions

As stated in this manual, when generating sparsity pattern or analytical derivative code, care must be taken when dimensioning variables within the root program unit. If an array is specified as an independent or dependent variable then the array must be dimensioned to its exact value in the root program unit. For example, suppose the original root program unit has the interface:

```
subroutine res(n,m,x,y),
```

where the first `n` entries of `x` are specified as the independent variables and the first `m` entries of `y` are specified as the dependent variables. Then `x` and `y` must be dimensioned using `n` and `m`, respectively. This is due to the fact that DAEPACK must know the dimensions of the independent and dependent variables for code generation. Fortunately, by using the symbolic dimensions (i.e., `n` and `m`), the same generated code can be used regardless of the actual values of `n` and `m` during run-time. The situation is a bit more complicated for intermediate active variables. Let z denote an intermediate variable (e.g., some program variable that holds intermediate results during the computation of the dependent variables from the independent variables). Variable z is defined as an *active variable* if it depends directly or indirectly on one ore more independent variables ***and*** one or more dependent variables depend directly or indirectly on z. (Fortunately, DAEPACK will determine these variables automatically so there is no need to search the source for them before generating code.) Problems arise when certain elements of array arguments of the root program unit are active. If such an array has a single dimension (e.g., `z(10), w(*)`) then the array must be dimensioned to a symbolic or numeric value equal to the largest index of any active element of the array. For example, if `w` is an array argument of the root program unit and `w(1000)` is the active element of `w` with the largest index (`1000`) then `w` must be dimensioned to at least `1000` in the root program unit. Since this information is in general not known a priori (e.g., the indices of the active array elements may not be known until the program is executed), these arrays should be dimensioned (symbolically or numerically) to the values actually used to allocate the arrays. The same situation holds for multidimensional arrays. In this case, the multidimensional array can be considered to be a one dimensional array indexed in column-major order (i.e., the entries in column 1 are numbered sequentially followed by the entries in column 2 and so on). Again, to be safe, it is better to dimension the multidimensional array arguments to the same values (symbolically or numerically) that they were dimensioned when allocated. DAEPACK will identify these variables prior to code generation and error messages will be issued if any problems are encountered.

To reiterate, in order to avoid unexpected behavior (and incorrect results), array arguments of the root program unit should be dimensioned to the values used to originally allocate them. (None of the array arguments of subroutines and functions called from the root program need to be dimensioned exactly.) To keep the code general (e.g., so that the same code can be used for different input sizes) the arrays should be dimensioned with integer variables that are passed in as arguments. For example, consider the following code segment from the root program unit of some problem:

```
      subroutine res(n,m,x,y,rpar1,rpar2,ipar)
!!independent { x }
!!dependent   { y }
      integer n,ipar(*)
      double precision x(*),y(*),rpar1(*),rpar2(*)
      .
      .
      .
```

where the user intends to generate code with `x(1:n)` being the independent variables and `y(1:n)` being the dependent variables. This is valid Fortran syntax, however, DAEPACK will not even generate code from this source because it cannot compute the number of independent and dependent variable and doesn't know the sizes of the active array arguments (in this example, assume `rpar1` and `rpar2` are active arrays). Suppose the user knows that `rpar1` and `rpar2` will never be larger than 1000 element arrays any time this subroutine is called. Then the following code segment will generate correct code:

```
      subroutine res(n,m,x,y,rpar1,rpar2,ipar)
!!independent { x }
```

```
!!dependent   { y }
      integer n,ipar(*)
      double precision x(n),y(m),rpar1(1000),rpar2(1000)
      .
      .
      .
```

Notice that `ipar` is still dimensioned with an asterisk. This is due to the fact that, in this example, the integer array does not contain any active elements. This is fine provided no element of `rpar1` or `rpar2` beyond element 1000 is ever accessed during a call to `res`. Often, this is not know a priori and, thus, a better alternative would be to modify the original code as follows:

```
      subroutine res(n,m,x,y,rpar1,len1,rpar2,len2,ipar)
!!independent { x }
!!dependent   { y }
      integer n,ipar(*),len1,len2
      double precision x(n),y(m),rpar1(len1),rpar2(len2)
      .
      .
      .
```

Now the user is forced to specify lengths for the workspace arrays and as long as these are set correctly, the generated code should be correct. Since DAEPACK requires more than what is syntactically correct, a form is displayed in the GUI just before code is generated. This form, shown in Figure 11 below, displays a list of all active array arguments of the root program unit and their specified dimensions. The user should examine these values to make sure they are correct. If not, the model should be closed and the file should be edited before any code is generated.

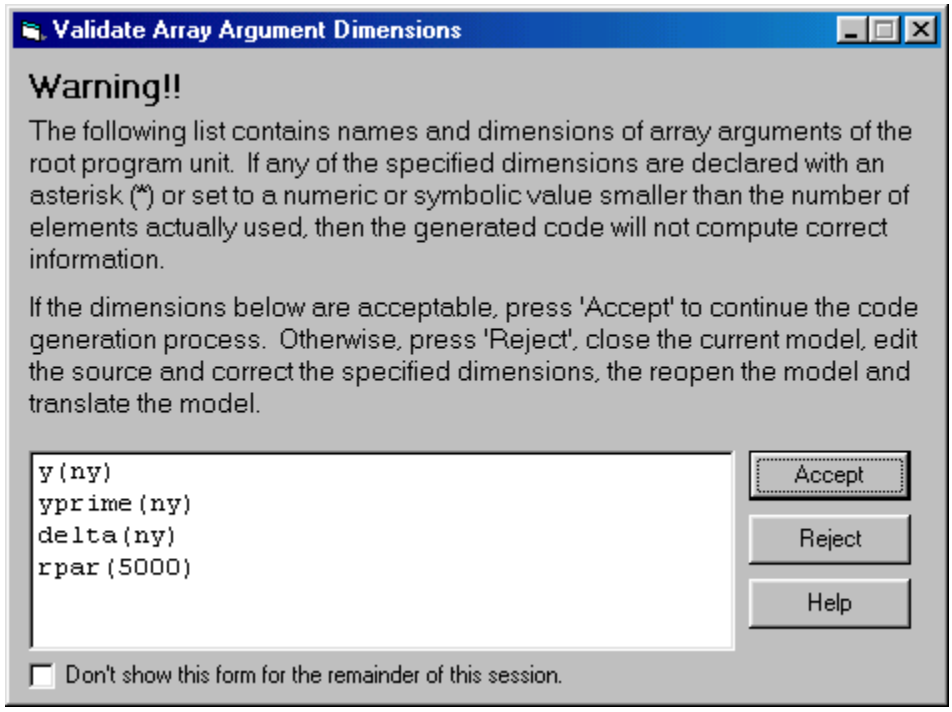


Figure 11. Array argument validation form.