DAEPACK

DSL48E Manual

Version 1.0

Numerical Integration with Robust State Event Location

John E. Tolsma

March 24, 2001

Contents

DAEPACK component DSL48E is an extension of DSL48S for the integration of differential/algebraic equations (DAEs) with discontinuities. DSL48E is called to take an integration step and upon successful completion, two outcomes are possible: no state events occur over the integration step taken, in which case DSL48E takes the largest step possible satisfying stability and accuracy, or a state event is detected, in which case, DSL48E takes a step to the event, adjusts the states and time derivatives so they are consistent at the event, and returns a flag indicating which state event has occurred.

This manual presents a brief description of hybrid discrete/continuous modeling and simulation, a description of the algorithms employed in DSL48E, and a description of what must be supplied by the user in order carry out the simulation. In addition, this manual describes how the additional information required for hybrid discrete/continuous simulation can be generated automatically with the DAEPACK symbolic components simply given the code for computing the DAE residuals.

1 Introduction

This manual describes the use of DAEPACK component DSL48E for performing hybrid discrete/continuous simulation. The following section provides background material on hybrid discrete/continuous simulation including a description of the formulation of the discrete aspects required when using DSL48E and a derivation of the steps required to perform state event location and state event polishing (i.e., determination of consistent states at the state event time) using the algorithm described in [5]. This is followed by a description of the DSL48E interface as well as a description of the callback functions that must be provided by the user of DSL48E to communicate the required information. Finally, the automatic generation of these callbacks from the user-supplied code computing the DAE residuals using the DAEPACK symbolic components is described.

2 Background

This section contains a description of the formulation of the hybrid discrete/continuous simulation required by DSL48E. The discussion is kept brief since more detailed descriptions can be found in [5, 1, 2, 3].

Most simulations of continuous systems exhibit a certain amount of discrete behavior. In the case of process simulation, this discrete behavior may be imposed on the system by the interaction with the safety interlock system or changes imposed on the operation such as the opening and closing of valves or the introduction of process disturbances. Discrete changes may also arise due to the choice in the modeling of the physico-chemical behavior of the system. For example the transition from laminar to turbulent flow as the Reynold's number passes through a critical value or the appearance of a second (thermodynamic) phase as the equilibrium temperature of a mixture rises above the bubble temperature or below the dew temperature. The overall simulation can be viewed as a collection of continuous simulation problems with the transition from one continuous problem to another occurring at *events* or *discontinuities*. One can distinguish between two types of events: time events[1] and state events. A time event is also referred to as an *explicit discontinuity* since when or where it occurs is known a priori. State events, which depend on the current state of the system, are also referred to as *implicit discontinuities* and when or if they occur during a simulation is not known a priori. A state event may result from the presence of an IF statement, such as,

```
IF (X(I)>=1.0e-4 AND Temp <= 500) ...
```

or, less obvious, through a nonsmooth intrinsic function, such as,

```
MAX(Z,10).
```

Time events are better processed by the simulation executive since they can be handled most efficiently by having the integrator stop at the time event. Consequently, state events are emphasized in this manual, however, both state and time events can be identified with DSL48E.

The logical expression associated with a state event will be referred to as a *state condition*. For example, the state conditions above are `X(I)>=1.0e-4 AND Temp <= 500` and `Z>=10`. As described in [5], a state condition is a logical proposition that contains a set of logical operators (e.g., `AND`, `OR`, `NOT`) and a set of relational expressions (containing relational operators $<$, $\leq$, $>$, and $\geq$). The value of the state condition will change (i.e., become active (true) or inactive (false)) at points where one or more of

[1] Here time refers simply to the independent variable we're integrating over.

the relational expressions become satisfied. For example, in the state condition above suppose `Temp` is less than `500` and `X(I)` is less than `1.0e-4`. The state condition value may change at the point where `X(I)` equals `1.0e-4` (provided `Temp` is still less than `500`). These relational expressions can be rearranged into the form:

$$g_i(w) \geq (>)\ 0 \tag{1}$$

where w is the set of variables and parameters contained in the original relational expression (for a DAE these may be time derivatives, differential variables, algebraic variables, time, etc.). The expression $g_i(x)$ is referred to as a *discontinuity function* and with the convention above, a positive value for g_i will be true and a negative value false. The value of the state condition may change at points where one or more discontinuity functions cross zero. Most modern state event location algorithms, including the one employed in DSL48E, rely on finding zero crossings of the discontinuity functions to identify the state events. Before using DSL48E, each state condition in the model must be identified, enumerated, and split into a set of discontinuity functions in the form shown in equation (1). For example, the state condition above is

```
X(I)>=1.0e-4 AND Temp <= 500
```

and the corresponding discontinuity functions are

```
X(I) - 1.0e-4 >= 0
```

and

```
500 - Temp >= 0.
```

Similarly, the implicit state condition

```
MAX(Z,10)
```

has the corresponding discontinuity function

```
Z - 10 >= 0.
```

DSL48E is used like DSL48S [7]: the subroutine is called repeatedly to perform a series of integration steps (in fact, DSL48S is called within DSL48E to perform the actual step). DSL48E must be called with the model *locked* into the current mode (i.e., `IF` statements as well as nonsmooth intrinsic functions must be fixed regardless of the value of the corresponding logical expressions). DSL48E will take the integration step and perform the state event location algorithm. One of the following will occur upon completion of the step:

1. the integration is advanced a single step containing no state events,
2. a state event occurs and the integration is advanced to the earliest state event time and consistent states and time derivatives at this point are returned, or
3. the integration fails for some reason.

If the integration step was successful and no events occurred, the integration is advanced by calling DSL48E again. If a state event was identified, the model is locked into the new mode (the state condition causing the state event is returned so that the user knows which mode is now active), a consistent re-initialization calculation is performed for the new model, and the integration is advanced on this new subdomain by calling DSL48E again (with the model locked in the new mode). If the integration fails, the user must take the appropriate action based on the error information returned (see also DSL48S documentation [7]).

The state event location algorithm employed by DSL48E consists of two main phases: state event detection/location and state event polishing. The first phase determines whether or not one or more events occur over the integration step just taken. If events occur, the event time is determined using the algorithm described in [5] and the earliest (and thus correct) state event time is guaranteed to be found. The only information the user must provide for this is the state conditions, the associated discontinuity functions, and a callback function that is used within DSL48E to query whether or not a state condition has changed (active to inactive or vice versa). How this information is passed to DSL48E is described in the following section.

The second phase is state event polishing. Here, the state event time determined in the previous phase is adjusted to prevent the phenomenon of *discontinuity sticking* [5] and consistent values for the states and time derivatives are determined at this new time. The remainder of this section describes the algorithms employed during this phase in order to put meaning behind the information and callbacks the user of DSL48E must provide.

As stated above, the state event location phase of the algorithm determines the earliest time an event occurs over a given step (provided one or more events occur). In the approach described in [5] and implemented in DSL48E, the state event location is performed by applying interval arithmetic [4] to the BDF interpolating polynomials corresponding to the discontinuity functions in order to find the zero crossings. Suppose that a state event occurs over a particular time step and the state event location phase indicates discontinuity function g^* crosses zero at a point t^*. Simply interpolating the polynomials associated with the states and time derivatives (also available) is not enough for the following reasons:

1. The interpolating polynomials do not guarantee that the states and time derivatives will be consistent with the DAE between mesh points,

2. There is no guarantee that the interpolated states and time derivatives will lie on the "correct side" of the discontinuity, causing the state condition to flip back immediately to its previous mode when the integration is restarted.

The second issue is referred to as discontinuity sticking. These two issues are addressed during the state event polishing phase.

Let

$$f(\dot{x}, x, y, u(t), t) = 0 \tag{2}$$

denote the original hybrid DAE currently being integrated, where $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} \times \mathbb{R}^{n_u} \times \mathbb{R} \longrightarrow \mathbb{R}^{n_x+n_y}$, x, $\dot{x}$ are the differential variables and their time derivatives and are function of time in an n_x-dimensional function space, y are the algebraic variables and are functions of time in an n_y-dimensional function space, $t \in \mathbb{R}$ is the integration variable, and $u(t)$ are the inputs and are functions of time in an n_u-dimensional function space. This DAE is augmented with the discontinuity functions as follows:

$$F(\dot{x}, x, y, \delta, u(t), t) = \begin{pmatrix} f(\dot{x}, x, y, u(t), t) \\ \delta - g(\dot{x}, x, y, u(t), t) \end{pmatrix} = 0 \tag{3}$$

where g are the set of discontinuity functions associated with the current discrete mode and δ are the *discontinuity variables*. The additional equations appended to the end of the original DAE are referred to as *discontinuity equations* and are present to place the discontinuity functions under integration error control. The discontinuity variables are purely algebraic. Consequently, the notation of the augmented system can be simplified by redefining the algebraic variables as

$$y \equiv \begin{pmatrix} y \\ \delta \end{pmatrix} \tag{4}$$

and increasing the value of n_y accordingly. Equation (3) is the form required by DSL48E. DSL48E assumes that

$$\begin{pmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial y} \end{pmatrix} \tag{5}$$

is regular (nonsingular) which is a sufficient condition for the DAE to be index 1. The following system is solved in order to determine consistent states and time derivatives at a time (close to t^*) which avoids discontinuity sticking:

$$F(\dot{x}, x, y, t, u(t)) = 0 \tag{6}$$

$$g^*(\dot{x}, x, t, u(t)) = \Delta g^* \tag{7}$$

$$x = x^p(t), \tag{8}$$

where g^* is the discontinuity function whose zero crossing has triggered the event, Δg^* is some sufficiently small positive or negative constant which places the new point on the correct side of the discontinuity (this value is automatically computed by DSL48E), and $x^p(t)$ are the interpolating polynomials associated with the BDF method for the differential variables. This system is $2n_x + n_y + 1$ equations in terms of $2n_x + n_y + 1$ variables ($\dot{x}$, x, y, and t). Equations (6)-(8) are solved using a quasi-Newton iteration scheme, solving for $\dot{x}$, y, and t (equation (8) is used to replace x with the explicit functions of time in equations (6) and (7)). An initial guess is obtained with the interpolation polynomials evaluated at the estimated event time. The following system,

$$\begin{bmatrix} \frac{\partial F}{\partial \dot{x}} & \frac{\partial F}{\partial y} & \frac{\partial F}{\partial t} + \frac{\partial F}{\partial x}\frac{dx^p}{dt} + \frac{\partial F}{\partial u}\frac{du}{dt} \\ \left(\frac{\partial g^*}{\partial \dot{x}}\right)^T & \left(\frac{\partial g^*}{\partial y}\right)^T & \frac{\partial g^*}{\partial t} + \left(\frac{\partial g^*}{\partial x}\right)^T \frac{dx^p}{dt} + \left(\frac{\partial g^*}{\partial u}\right)^T \frac{du}{dt} \end{bmatrix} \begin{bmatrix} \delta\dot{x} \\ \delta y \\ \delta t \end{bmatrix} = \begin{bmatrix} F \\ g^* - \Delta g^* \end{bmatrix}$$

$$\begin{bmatrix} A & d \\ c^T & \alpha \end{bmatrix} \begin{bmatrix} \delta\dot{x} \\ \delta y \\ \delta t \end{bmatrix} = \begin{bmatrix} b \\ \beta \end{bmatrix}, \tag{9}$$

is solved for Newton steps $\delta\dot{x}$, δy, and δt. Note that only the $(n+m) \times (n+m)$ matrix A in equation (9) needs to be factored. The quasi-Newton iteration scheme will attempt to converge this system evaluating and factoring this matrix only once (because the interpolated solution should be sufficiently close to the actual solution). The additional vector d and scalar α must be provided by the user of DSL48E through callbacks (vector c can be computed with the information already available). This iteration will (hopefully) converge to a consistent set of states and time derivatives at a state event time that avoids discontinuity sticking. These consistent values and time will be returned to the user along with an indication of which state condition has changed so that the user may re-initialize the system in the new mode and continue the integration. The following section describes the information that must be provided by the user in order to use DSL48E.

3 Using DSL48E

As described above DSL48E is called to take an integration step. The interface to DSL48E is shown below.

```
      subroutine dsl48e(res, senrhs, neq, time, z, zdot, tout, info,
     1   rtol, atol, idid, rwork, lrw, iwork, liw, rpar, ipar, jac,
     2   nejac, jrow, jcol, jydot, njydot, jdtype, bounds,
c.....the following arguments are for state event location
     3   ievent, idiscon, nc, nd, isc, tevt, lievwk, ievwk,
     4   lrevwk, revwk, stchange, evtderiv)
c### 990804 j.e.tolsma copyright mit
c### dsl4se - dsl48s with state event location (no sensitivities).
c###
c### description:
c###
c### arguments: (# indicates argument must be set prior to call)
c###
c### the first 25 arguments are identical to the arguments of
c### of dsl48s.  see the dsl48s documentation for a description
c### of these. the following list describes the arguments
c### specific to state event location.
c###
c###     ievent    - flag indicating whether or not an event
c###                 has occurred over the previous time step.
c###                 ievent = 0 - no event
c###                        = k - state event k occurred during
c###                              previous step.
c###     idiscon   - if ievent <> 0 then idiscon is the discontinuity
c###                 function is state event ievent that triggered the
c###                 discontinuity.
c### (#) nc         - number of state events
c### (#) nd         - total number of discontinuity functions
c### (#) isc(nc)    - number of discontinuity functions associated
c###                 with each event, isc(k) equals the number of
c###                 discontinuity functions associated with state
c###                 event k.
c###     tevt      - time of earliest event over previous time step
c###                 if one has occurred
c### (#) lievwk     - length of integer workspace for event location
c###     ievwk     - integer workspace for event location
c### (#) lrevwk     - length of real workspace for event location
c###     revwk     - real workspace for event location
c### (#) stchange  - user supplied subroutine (see below)
c### (#) evetderiv - user supplied subroutine (see below)
c###
```

The first 25 arguments are the same as DSL48S and are described in the accompanying documentation [7]. However, the residual and Jacobian subroutines, `res` and `jac`, must perform discontinuity-locked evaluations of the augmented model (see below). As stated above, the original hybrid DAE model equations, f, must be augmented as follows:

$$f(\dot{z}, z, t) = 0 \tag{10}$$

$$z_{neq-n_d+i} - g_i(\dot{z}, z, t) = 0 \qquad i = 1, \ldots, n_d \tag{11}$$

where $neq = n_x + n_y$, $z = (x, y)$, and $\{g_i\}_{i=1}^{n_d}$ are the discontinuity functions associated with the current set of state conditions. Note that the controls have been removed from the argument list. This is possible since they are explicit functions of time. The discontinuity equations (11) and variables are appended to the original DAE model and state variable vector in the following order: all of the discontinuity functions associated with state condition 1 are followed by the discontinuity functions associated with state condition 2 and so on. Furthermore, the additional discontinuity variables **must** be in the last `nd` entries of z and the discontinuity equations **must** be in the last `nd` entries of the residual vector.

The current implementation of DSL48E does not permit parametric sensitivity calculations to be performed beyond the first state event. However, this extension is currently under development. The remainder of the arguments, specific to DSL48E, are described below.

Argument `ievent` is an output integer variable that equals zero if no event has occurred over the step just taken or equals `k` if state event `k` has occurred. If `ievent` is nonzero then the states and time derivatives, `z` and `zdot`, will be consistent at the state event time returned in `tevt` (provided the state event location and polishing algorithms do not fail) and the index of the discontinuity function is returned in `idiscon`. Argument `nc` is an input integer variable containing the number of state conditions currently present in the model. Argument `nd` is an input integer variable containing the total number of discontinuity functions currently present in the model. Integer array argument `isc(nc)` specifies the number of discontinuity functions associated with each state condition (i.e., state event `k` contains `isc(k)` discontinuity functions). Argument `tevt` is an output double precision variable returning the earliest event time if `ievent` is nonzero, otherwise its value is undefined. Integer and double precision array arguments `ievwk(lievwk)` and `revwk(lrevwk)` are used as workspace. The contents of `ievwk` and `revwk` between steps is undefined. The size of the integer workspace, `lievwk`, must be greater than or equal to the number of differential variables (dimension of x) plus the maximum number of discontinuity functions in any of the state conditions (i.e., the maximum entry in `isc(1:nc)`), or `neq`, whichever is larger. The size of the double precision workspace must be greater than or equal to `(q+1)*nd + q + 5*neq + 40` where `q` is the maximum order of the integration (contained in `info(9)` which has a default value of 5). The two remaining arguments, described below, are user-supplied callback functions. Table 1 contains a summary of the additional arguments required by DSL48E.

The additional information required by the event location algorithm is provided by the user through the use of callbacks. The first subroutine required is `stchange` with interface shown below.

```
      subroutine stchange(ichange,istate,neq,nc,nd,isc,
     1   t,z,zdot,rpar,ipar,ndi,ig)
```

This subroutine is used by DSL48E to determine whether or not a given state condition has flipped. The argument `ichange` is an output integer variable that should be set equal to one if state condition `istate` has changed or zero otherwise. Argument `istate` is an input integer variable that is equal to the index of the state condition that is currently being examined (recall that the state conditions must be enumerated). As above, `neq`, `nc`, `nd` are the total number of DAE states and discontinuity variables,

Table 1: Summary of additional arguments required by DSL48E. All other arguments are the same as DSL48S (except that the residual and Jacobian routines perform locked evaluations).

Status	Type	Name	Description
output	integer	`ievent`	Flag indicating whether or not an event has occurred over the previous time step (nonzero value indicates which event has occurred).
output	integer	`idiscon`	If `ievent` is nonzero, this argument contains the index of the discontinuity function that triggered the event.
input	integer	`nc`	Number of state events.
input	integer	`nd`	Total number of discontinuity functions.
input	integer	`isc(nc)`	Number of discontinuity functions associated with with each event, `isc(k)` equals the number of discontinuity functions associated with state event `k`.
output	double	`tevt`	Time of earliest event over previous time step if one has occurred.
input	integer	`lievwk`	Length of integer workspace for event location.
—	integer	`ievwk(lievwk)`	Integer workspace for event location.
input	integer	`lrevwk`	length of real workspace for event location.
—	double	`revwk(lrevwk)`	Real workspace for event location.
input	subroutine	`stchange`	User-defined subroutine indicating whether or not a state condition has changed.
input	subroutine	`evetderiv`	User supplied subroutine returning partial derivatives required for state event polishing.

total number of state conditions, and total number of discontinuity functions, respectively. Arguments `time`, `z(neq)`, `zdot(neq)`, `rpar`, and `ipar` contain the current values of time, DAE states and time derivatives, and user-supplied parameters that may be required to determine whether or not the state condition has changed. Integer array `ig(ndi)` is an input variable containing the current values of the discontinuity functions associated with state condition `istate` with the following convention: `ig(k)` is zero if discontinuity function `k` is false (i.e., less than zero) and one if it is true (i.e., greater than or equal to zero). **Variable `ichange` should be the only argument modified.** The arguments are summarized in Table 2.

The second subroutine that must be provided by the user is `evtderiv` with interface:

```
      subroutine evtderiv(icode,istate,idiscon,neq,nc,nd,isc,
     1   time,z,zdot,rpar,ipar,d,alpha)
```

which provides the additional derivatives required for state event polishing (i.e., vector d and scalar α in equation (9)). Argument `icode` is an output integer variable that is equal to zero upon returning from `evtderiv` if the calculations were successful, nonzero otherwise. Input integer arguments `istate` and `idiscon` specify the discontinuity function of interest (g^* in equation (7)); g^* is the `idiscon`-th

Table 2: Summary of arguments for user-supplied callback function `stchange`. This subroutine is used to indicate whether or not a state condition has changed.

Status	Type	Name	Description
output	integer	`ichange`	Equals one if state event `istate` has changed or zero, otherwise.
input	integer	`istate`	Index of the state condition of interest.
input	integer	`neq`	Total number of DAE states and discontinuity variables.
input	integer	`nc`	Total number of state conditions.
input	integer	`nd`	Total number of discontinuity functions.
input	integer	`isc(nc)`	Number of discontinuity functions associated with with each event, `isc(k)` equals the number of discontinuity functions associated with state event `k`.
input	double	`time`	Current value of independent variable.
input	double	`z(neq)`	Current values of DAE states and discontinuity variables.
input	double	`zdot(neq)`	Current values of time derivatives of the DAE states and discontinuity variables.
input	double	`rpar(*)`	User-supplied real parameters passed to residual and Jacobian subroutines.
input	integer	`ipar(*)`	User-supplied integer parameters passed to residual and Jacobian subroutines.
input	integer	`ndi`	Number of discontinuity functions associated with state condition `istate`.
input	integer	`ig(ndi)`	Array containing the current values of the discontinuity functions associated with state condition `istate`, `ig(k)` $= 1$ if discontinuity function is true (non-negative), 0,otherwise.

discontinuity function of the `istate`-th state condition. Input arguments `neq`, `nc`, `nd`, and `isc` are the same as described above. Input variables `time`, `z` and `zdot` are the point at which the derivatives are to be evaluated (`rpar` and `ipar` are the same arrays that are passed to the residual and Jacobian routines). Output arguments `d` and `alpha` should be set equal to:

$$d = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial u}\frac{du}{dt} \tag{12}$$

$$\alpha = \frac{\partial g^*}{\partial t} + \left(\frac{\partial g^*}{\partial u}\right)^T \frac{du}{dt} \tag{13}$$

where F is the augmented DAE model (equations (10) through (11)). Note that these are ***not*** the same as the d an α defined in equation (9). The additional terms in these arrays as well as the vector c are constructed internally with the information returned in the user-supplied Jacobian routine, `jac`. **The only arguments that should be modified are `icode`, `d`, and `alpha`.** A summary of the arguments of `evtderiv` is contained in Table 3.

Table 3: Summary of arguments for user-supplied callback function `evtderivs`. This subroutine returns the additional partial derivatives required for state event polishing.

Status	Type	Name	Description
output	integer	`icode`	Error flag. Set equal to a nonzero value if an error has occurred.
input	integer	`istate`	Index of the state condition of interest.
input	integer	`idiscon`	Index of the discontinuity function of the state condition of interest.
input	integer	`neq`	Total number of DAE states and discontinuity variables.
input	integer	`nc`	Total number of state conditions.
input	integer	`nd`	Total number of discontinuity functions.
input	integer	`isc(nc)`	Number of discontinuity functions associated with with each event, `isc(k)` equals the number of discontinuity functions associated with state event `k`.
input	double	`time`	Current independent variable the derivatives are to be evaluated at.
input	double	`z(neq)`	Current values of DAE states the derivatives are to be evaluated at.
input	double	`zdot(neq)`	Current values of time derivatives of the DAE states the derivatives are to be evaluated at.
input	double	`rpar(*)`	User-supplied real parameters passed to residual and Jacobian subroutines.
input	integer	`ipar(*)`	User-supplied integer parameters passed to residual and Jacobian subroutines.
output	double	`d(neq)`	Computed partial derivatives (see equation (12))
output	double	`alpha`	Computed partial derivatives (see equation (13))

4 Generating the Additional Information with DAEPACK

A substantial amount of information is required to perform numerical integration of a hybrid discrete/continuous model correctly. Not only must the user code a set of one or more subroutines returning the residuals of the DAE but these subroutines must be able to perform locked evaluations, extract the discontinuity functions, and return the residuals of the augmented model. There must also be a subroutine that determines whether or not a given state condition has flipped, a subroutine that returns the partial derivatives of the augmented model with respect to $\dot{z}$ and z, and a subroutine that returns the partial derivatives with respect to time (for vector d and scalar α). The difficulty in generating this additional information may make the use of DSL48E prohibitively difficult for large, complicated models. This is particularly true when legacy code is used. Fortunately, the symbolic components of DAEPACK [6] can be used to generate ***all*** of the additional information automatically. All the user must provide is a subroutine returning the residuals of the original hybrid DAE model.

Figure 1 contains a diagram showing the code generated automatically by DAEPACK. In this example,

the user provides a single subroutine, named `RES0`, which returns the residuals of the hybrid DAE. This subroutine may be arbitrarily complex and call any number of additional subroutines and functions (however, source code must be available for all subroutines and non-intrinsic functions used). A makefile is provided with the DAEPACK Symbolic Library distribution that is used to generate automatically all of the necessary code and create a shared library that can be linked with the application using DSL48E. In addition, the DAEPACK DSL48E distribution contains an template program file, named `ds148e_main.f`, that shows how DSL48E can be used to perform a dynamic simulation, including reinitialization at the event. This file can be edited for a particular problem or used as a reference for incorporating DSL48E into a larger application.

5 Distribution

DAEPACK component DSL48E is currently available within the DAEPACK libraries

`libdaepack_<`*sparse*`>_<`*platform*`>_<`*version*`>.a`

and

`libdaepack_<`*dense*`>_<`*platform*`>_<`*version*`>.a`

respectively, for UNIX and Linux, where *platform* is the platform for which the library is compiled and *version* is the version number, and *sparse* and *dense* refer to which linear algebra packages are used. See the DAEPACK webpage (http://yoric.mit.edu/daepack) for available platforms and versions. The sparse version of the libary contains code that exploits sparsity during linear algebra calculations using the Harwell MA48 routines. Consequently, the user must provide this additional code before using DSL48E. A complete description of which Harwell routines is given in [8]. The dense linear algebra versions of the library are identical to those described here except for the fact that dense linear algebra is performed and the Harwell routines need not be present. The interfaces to all of the code within the sparse and dense versions of the library are the same. Hence, if at some later time the Harwell routines are available, you must simply recompile the application, linking in the sparse versions of the libraries and the Harwell code.

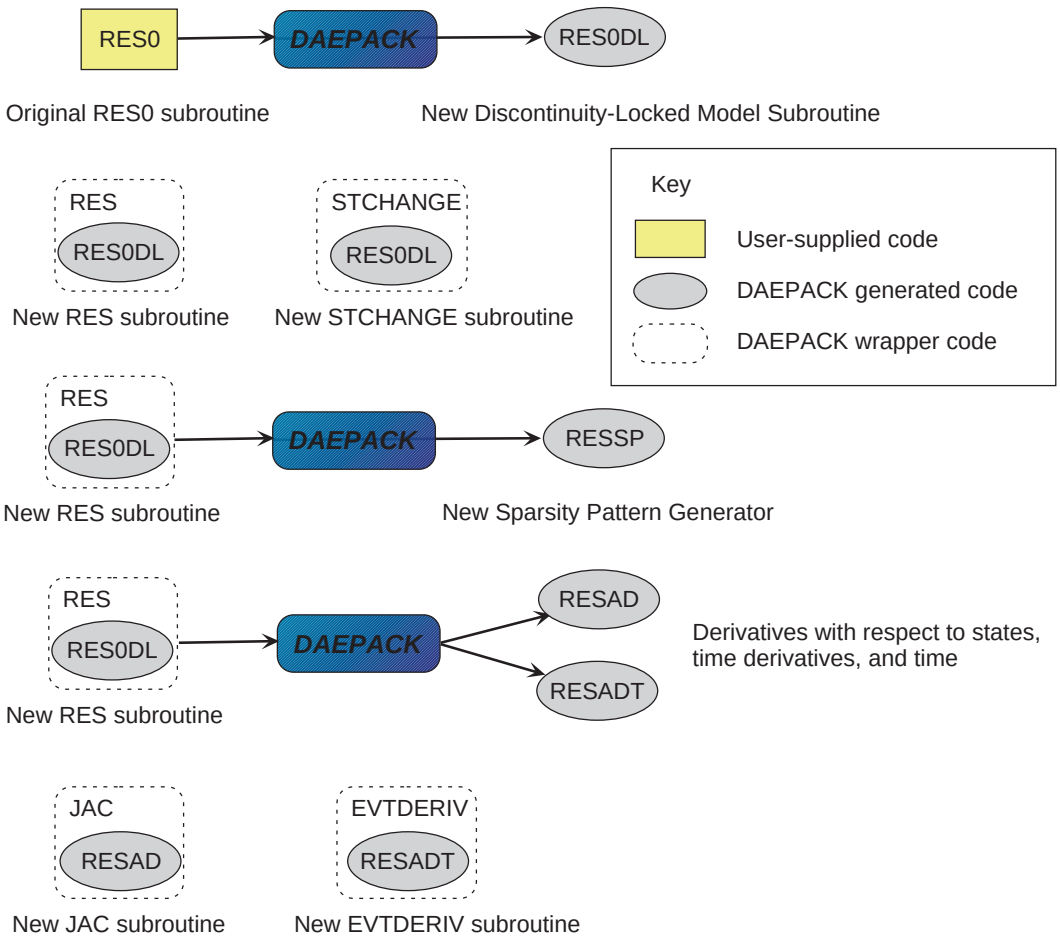

Figure 1: Automatic generation of additional information required for hybrid dynamic simulation with DAEPACK.

Appendix A

The Discontinuity-locked Model

DAEPACK components DSL48E and DSL48SE require a *discontinuity-locked* version of the code evaluating the DAE of interest. If the code of a model contains discontinuities (e.g., IF statements and nonsmooth intrinsic functions) then a given set of inputs define a particular branching through the code. For example, consider the pseudo-code shown below:

```
statement 1
statement 2
if (x1 > 0) then
   statement 3a
else
   statement 3b
end
if (x2 < 0) then
   statement 4a
else
   statement 4b
end
```

If this code is executed with both `x1` and `x2` greater than zero then the sequence of statements encountered is:

```
statement 1
statement 2
statement 3a
statement 4b
```

In a discontinuity-locked model, the conditional branching through the code is fixed to a particular mode and the same sequence of statements are executed regardless of the values of the input. By fixing the conditional branching, a smooth model is always evaluated and any changes in conditional branching is handled explicitly by DSL48E or DSL48SE.

The DAEPACK components DSL48E and DSL48SE do not place a restriction on how this discontinuity-locking is achieved. One option is to use the code generation capabilities of DAEPACK to automatically construct the discontinuity-locked model and extract the discontinuity functions that trigger changes in discrete modes. How this is actually achieved is beyond the scope of this paper and the user does not need to know these details in order to use DSL48E and DSL48SE. The remainder of this section describes the interface of the code generated by DAEPACK for performing locked evaluations.

Suppose the original DAE subroutine (that is, the one with hidden discontinuities) has the following interface:

```
subroutine res0(neq,t,y,ydot,delta,ires,ichvar,rpar,ipar)
implicit none
integer neq,ires,ichvar,ipar(*)
double precision t,y(neq),ydot(neq),delta(neq),rpar(*)
```

```
      .
      .
      .
```

This is the same as the one provided in the example above. The discontinuity-locked model constructed automatically by DAEPACK has the following augmented interface:

```
      subroutine res0(neq,t,y,ydot,delta,ires,ichvar,rpar,ipar,
     1   nsc,ndf,isc,iscchng,discon,locked,istate,ig)
      implicit none
      integer neq,ires,ichvar,ipar(*)
      double precision t,y(neq),ydot(neq),delta(neq),rpar(*)
      integer nsc,ndf,isc(nsc),iscchng(nsc),istate,ig(ndf)
      double precision discon(ndf)
      logical locked
      .
      .
      .
```

The interface to the code generated by DAEPACK is the same as the original with additional arguments appended. Integer arguments `nsc` and `ndf` are the total number of state conditions and discontinuity functions in the current discrete mode and are computed during the call to `res0dl`. Integer array `isc(1:nsc)`, also set in this subroutine, has entries equal to the number of discontinuity functions contained in each of the `nsc` state conditions. For example, state condition `k` is composed of `isc(k)` discontinuity functions. Variable `ndf` equals the sum of the `nsc` entries of `isc`. Integer array `iscchng(1:nsc)` is also an output variable. The value of `iscchng(k)` will be nonzero if the value of state condition `k` has changed since the previous call to the discontinuity locked model. This array is useful for state conditions that are not decomposed into discontinuity functions (e.g., `i.eq.4`). Double precision array `discon(ndf)` contains the values of the discontinuity functions upon returning from `res0dl`. Logical argument `locked` is an input variable. If `locked` is set to `.false.` then the model is evaluated in an un-locked manner (i.e., the conditional branching is determined by the values of the subroutine inputs). During the model evaluation, the conditional branching is recorded within the generated code. On subsequent calls to `res0dl` with `locked` equal to `.true.` this same conditional branching is followed regardless of the values of the subroutine inputs. Variable `istate` is an input/output and array `ig(1:ndf)` is an input. When `res0dl` is called with `istate` not equal to zero then `ig(1:ndf)` must be initialized with the logical values of the discontinuity functions that are contained in the `istate`-th state condition. For example, if `ig(k)` is zero then the `k`-th discontinuity function of state condition `istate` is false. Similarly, if `ig(k)` is unity then the `k`-th discontinuity function of state condition `istate` is true. Using this information, the generated code will determine whether or not the `istate`-th state condition changes from its previous value with the logical values of the discontinuity functions contained in `ig`. If the state condition has changed value then a nonzero value is returned in `istate` otherwise `istate` is set to zero.

If DAEPACK is not used to generate the discontinuity-locked code then the user must be able to lock the model into a particular discrete mode until DSL48E or DSL48SE return and indicate an event has occurred. The user must then change the model to the new discrete mode, reinitialize the system, and continue the integration or sensitivity analysis.

References

[1] A. Back, J. Guckenheimer, and M. Meyers, *A dynamical simulation facility for hybrid systems*, in Hybrid Systems, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds., vol. 736 of Lecture Notes in Computer Science, New York, 1993, Springer-Verlag.

[2] P. I. Barton, *The Modeling and Simulation of Combined Discrete/Continuous Processes*, PhD thesis, University of London, London, U.K., May 1992.

[3] P. I. Barton, *Modeling, simulation and sensitivity analysis of hybrid systems*, in Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design, I. C. S. Society, ed., 2000.

[4] R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979.

[5] T. Park and P. I. Barton, *State event location in differential algegraic models*, ACM Transactions on Modelling and Computer Simulation, 6 (1996), pp. 137–165.

[6] J. E. Tolsma, *DAEPACK code generation manual*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.

[7] ——, *Large-scale numerical integration and parametric sensitivity analysis of DAEs. DSL48S manual*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.

[8] ——, *Supporting libraries for the DAEPACK numerical components*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.