DAEPACK

DSL48SE Manual

Version 1.0

Hybrid Discrete/Continuous Numerical Integration and Parametric Sensitivity Analysis

John E. Tolsma

March 24, 2001

Contents

DAEPACK component DSL48SE is an extension of DSL48E and DSL48S for the numerical integration and parametric sensitivity analysis of differential/algebraic equations (DAEs) with discontinuities.

This manual presents a description of hybrid discrete/continuous modeling, simulation and parametric sensitivity analysis, a description of the algorithms employed in DSL48SE, and a description of what must be supplied by the user in order carry out the sensitivity analysis. 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.

Packaged with the DAEPACK distribution is a collection of example codes illustrating how the numeric components may be used to solve a variety of basic problems (e.g., a simple numerical integration, solution of a set of nonlinear algebraic equations, and in the case of DSL48SE, a simply hybrid parametric sensitivity analysis). Section 5 contains a description of the code provided which uses DSL48SE. Users wanting to quickly use DSL48SE may go directly to this section to learn how to immediately begin using the code. The examples provided may also be used as a starting point for performing hybrid sensitivity analysis with other problems.

1 Introduction

This manual describes the use of DAEPACK component DSL48SE for performing numerical integration and parametric sensitivity analysis with hybrid discrete/continuous systems. The following section provides background material on hybrid discrete/continuous DAE models including a description of the formulation of the discrete aspects required when using DSL48SE and a derivation of the steps required to perform state event location, state event polishing (i.e., determination of consistent states at the state event time) using the algorithm described in [9], and computation of the sensitivity jump using the theory developed in [6]. This is followed by a description of the DSL48SE interface as well as a description of the callback functions that must be provided by the user of DSL48SE to communicate the required information. In Section 4, the automatic generation of these callbacks from the user-supplied code computing the DAE residuals using the DAEPACK symbolic components is described. Finally, Section 5 describes how to use the example code provided in the DAEPACK distribution example directories.

This manual supplements the DAEPACK DSL48S documentation [11]. The DSL48S documentation describes how to use DSL48S to perform numerical integration and parametric sensitivity analysis with smooth systems. DSL48SE extends DSL48S to handle nonsmooth models, however, the reader should be aware of the details of DSL48S since this code is called within DSL48SE. Hence, the DSL48S documentation should be familiar before continuing here.

A related DAEPACK component is DSL48E [12]. DLS48E extends DSL48S by allowing the user to perform numerical integration of nonsmooth systems. This component may be used instead of DSL48SE if sensitivities are not desired.

2 Background

This section contains a description of the formulation of the hybrid discrete/continuous model required by DSL48SE and a description of hybrid discrete/continuous parametric sensitivity analysis. These discussions are kept brief since more detailed descriptions can be found in [9, 1, 2, 3, 5, 6].

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 choices made 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 transitions 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. An event may result from the presence of an IF statement, such as,

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

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

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

```
Y = MAX(Z,10) + ....
```

In the case of hybrid discrete/continuous numerical integration, time events are better processed by the calculation executive since they can be handled most efficiently by having the integrator stop at the time event. However, in the case of parametric sensitivity analysis, the jump in sensitivities (explained below) caused by the time events must be explicitly computed. DSL48SE provides the capability of explicitly computing this jump and thus, time events and state events should be both handled by DSL48SE.

The logical proposition 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 [9], 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 may change (i.e., become active (true) or inactive (false)) at points in time where one or more of 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 will 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:

$$g_i(w, t; p) \geq \ (>)\ 0 \tag{1}$$

where w is the set of variables contained in the original relational expression (for a DAE these may be time derivatives, differential variables, algebraic variables, or inputs), t is time (the independent variable), and p is a set of time-invariant parameters. The expression $g_i(w, t; p)$ 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 in time where one or more discontinuity functions cross zero. Most modern state event location algorithms, including the one employed in DSL48SE, rely on finding zero crossings of the discontinuity functions to identify the state events. Before using DSL48SE, 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
```

and

```
500 - Temp.
```

Similarly, the implicit state condition

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

has the corresponding discontinuity function

```
Z - 10.
```

(Section 4 describes how this task, among others, is performed automatically with the code generation capabilites of DAEPACK.)

The discussion above describes how the appearance of IF statements and intrinsic functions such as MIN, MAX, and ABS in a code can introduce discontinuities into an otherwise smooth model. If a

standard numerical integration code (i.e., one that does not handle events explicitly) is used to integrate a model containing discontinuities then at the very least, in many cases, the calculation will be inefficient. However, often the calculation will fail or produce incorrect results. If only numerical integration is required then DAEPACK component DSL48E may be used to handle these events and perform properly the hybrid discrete/continuous simulation. If, however, parametric sensitivities are desired then using a code that does not explicitly handle the events will generally result in the computation of incorrect results without any indication of errors to the user [14]. To illustrate this, consider the following DAE:

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

where $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} \times \mathbb{R}^{n_u} \times \mathbb{R} \times \mathbb{R}^{n_p} \longrightarrow \mathbb{R}^{n_x+n_y}$, x, $\dot{x}$ are the differential state variables and their time derivatives, and are function of time in an n_x-dimensional function space, y are the algebraic state variables and are functions of time in an n_y-dimensional function space, $t \in \mathbb{R}$ is the integration variable, $u(t)$ are the inputs and are functions of time in an n_u-dimensional function space, and $p \in \mathbb{R}^{n_p}$ are the time invariant parameters. Parametric sensitivity analysis involves solving the DAE above along with the associated sensitivity system:

$$\frac{\partial f}{\partial \dot{x}}\frac{\partial \dot{x}}{\partial p_i} + \frac{\partial f}{\partial x}\frac{\partial x}{\partial p_i} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial p_i} + \frac{\partial f}{\partial p_i} = 0 \qquad i = 1, \ldots, n_p \tag{3}$$

Performing a parametric sensitivity analysis consists of numerically integrating the original DAE augmented with the sensitivity system to obtain trajectories for the states, $x(t;p)$ and $y(t;p)$, time derivatives, $\dot{x}(t;p)$, and sensitivities, $\dot{s}_x(t;p) \equiv \partial\dot{x}/\partial p$, $s_x(t;p) \equiv \partial x/\partial p$, and $s_y(t;p) \equiv \partial y/\partial p$. Several algorithms have been developed to solve this problem efficiently by exploiting structural similarities between the original DAE and sensitivity system [5, 7]. The occurrence of state events during the simulation often implies jumps in the sensitivity trajectories, even if the states are continuous. DSL48SE extends DSL48S by performing the additional tasks of correctly identifying the event and computing the resulting sensitivity jump. Failure to explicitly compute the jump at the events will result in the computation of incorrect sensitivities [14]. To illustrate this, consider the small hybrid ODE below:

$$\dot{x} = \begin{cases} 4 - x & : \quad x^3 - 5x^2 + 7x \leq p \\ 10 - 2x & : \quad \text{otherwise} \end{cases} \tag{4}$$

where $p = 2.9$ and $x(0) = 0$. Figure 1 contains the state and sensitivity trajectories for the example above. In one case, the discontinuity is hidden from the integrator and the results suggest that the state variable is not at all sensitive to the parameter. The other sensitivity was obtained with DSL48SE is correct.

DSL48SE is used like DSL48S [11]: the subroutine is called repeatedly to perform a series of integration steps to compute the state and sensitivity trajectories (in fact, DSL48S is called within DSL48SE to perform the actual step). Like DSL48E [12], DSL48SE must be called with the model *locked* into the current mode (i.e., `IF` statements as well as nonsmooth intrinsic functions must be locked into a single clause regardless of the value of the corresponding logical expressions). Discontinuity locking is described in more detail in Appendix A. DSL48SE 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. an event occurs and the integration is advanced to the earliest state event time and consistent states, time derivatives, and sensitivities at this point are returned, or

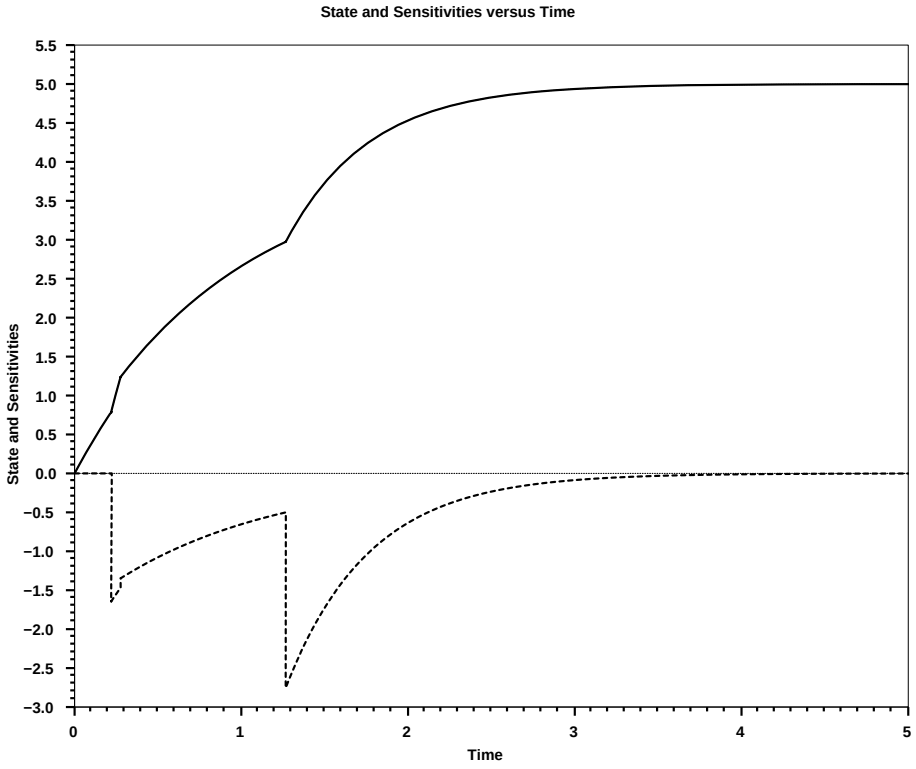

Figure 1: State and sensitivity trajectories small hybrid ODE. The heavy solid line is the state trajectory, the thin dotted line is the sensitivity trajectory computed with hidden discontinuities, and the dashed line exhibiting jumps is the true sensitivity trajectory.

3. the calculation fails for some reason.

If the integration step was successful and no events occurred, the calculation is advanced by calling DSL48SE again. If an event was identified, the user should lock the model into the new mode (the state condition causing a state event is returned so that the user knows which mode is now active), perform a consistent re-initialization calculation for the new model to compute values for the states and time derivatives at the beginning of the new mode, and then integration is advanced in this new mode by calling DSL48SE again. During the first call to DSL48SE after an event is located, the jump in sensitivities will be computed automatically. If the calculation fails, the user must take the appropriate action based on the error information returned (see also DSL48S documentation [11]).

The state event location algorithm employed by DSL48SE consists of three main phases: state event detection/location, state event polishing, and computation of the jump in sensitivities. The first phase, event detection, is performed during each integration step. The second phase is performed during a step in which an event has been detected and located. The third phase is performed on the subsequent call to DSL48SE after the second phase was performed successfully.

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 [9] 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 DSL48SE to query whether or not a state condition has changed value (active to inactive or vice versa). How this information is passed to DSL48SE 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* [9] and consistent values for the states, time derivatives, and sensitivities are determined at this new time.

The third phase computes the jump in sensitivities caused by the state or time event. The theory developed in [6] provides the equations required to determine this jump. The remainder of this section describes the equations that are solved during the event polishing phase and subsequent computation of the jump.

As stated above, the state event location phase of the algorithm determines the earliest time an event occurs over a given time step (provided one or more events occur). In the approach described in [9] and implemented in DSL48SE, the state event location is performed by applying interval arithmetic [8] 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.

The DAE (2) above is augmented with the discontinuity functions as follows:

$$F(\dot{x}, x, y, \delta, u(t), t; p) = \begin{pmatrix} f(\dot{x}, x, y, u(t), t; p) \\ \delta - g(\dot{x}, x, y, u(t), t; p) \end{pmatrix} = 0 \tag{5}$$

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

$$y \equiv \begin{pmatrix} y \\ \delta \end{pmatrix} \tag{6}$$

and increasing the value of n_y accordingly. Equation (5) is the form required by DSL48SE. DSL48SE assumes that

$$\begin{pmatrix} \frac{\partial F}{\partial \dot{x}} & \frac{\partial F}{\partial y} \end{pmatrix} \tag{7}$$

is regular (nonsingular), which is a sufficient condition for the augmented DAE to be index 1. System (5) will be integrated with DSL48SE. Although larger than the original DAE, very little additional work is required to solve this system due to the structure of the discontinuity equations [9]. Furthermore, the time trajectories of the discontinuity variables and their associated sensitivities obtained with DSL48SE provide useful insights into the model.

After an event is identified, the following system is solved in order to determine consistent states and time derivatives at a time (close to t^*) which avoids discontinuity sticking:

$$F(\dot{x}, x, y, u(t), t; p) = 0 \tag{8}$$

$$g^*(\dot{x}, x, u(t), t; p) = \Delta g^* \tag{9}$$

$$x = x^p(t), \tag{10}$$

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 DSL48SE), 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 (8)-(10) are solved using a quasi-Newton iteration scheme, solving for $\dot{x}$, y, and t (equation (10) is used to replace x with the explicit functions of time in equations (8) and (9)). An initial guess is obtained with the interpolation polynomials evaluated at the estimated event time. The following system,

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

$$\begin{bmatrix} A & d \\ c^T & \alpha \end{bmatrix} \begin{bmatrix} \delta\dot{x} \\ \delta y \\ \delta t \end{bmatrix} = \begin{bmatrix} b \\ \beta \end{bmatrix}, \tag{11}$$

is solved for Newton steps $\delta\dot{x}$, δy, and δt. Note that only the $(n+m) \times (n+m)$ matrix A in equation (11) 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 DSL48SE 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. At this point, the sensitivities will be interpolated at the event time. The sensitivity of the event time with respect to the parameters, $\partial t/\partial p$, is then computed by solving the following set of n_p linear equations:

$$\left(\frac{\partial g^*}{\partial \dot{x}^{(k)}}\right)^T \left(\dot{s}_x^{(k)} + \ddot{x}^{(k)}\frac{\partial t}{\partial p}\right) + \left(\frac{\partial g^*}{\partial x^{(k)}}\right)^T \left(s_x^{(k)} + \dot{x}^{(k)}\frac{\partial t}{\partial p}\right) + \left(\frac{\partial g^*}{\partial y^{(k)}}\right)^T \left(s_y^{(k)} + \dot{y}^{(k)}\frac{\partial t}{\partial p}\right) + \left(\frac{\partial g^*}{\partial u^{(k)}}\right)^T \left(\frac{\partial u^{(k)}}{\partial p} + \frac{\partial u^{(k)}}{\partial t}\frac{\partial t}{\partial p}\right) + \frac{\partial g^*}{\partial p} + \frac{\partial g^*}{\partial t}\frac{\partial t}{\partial p} = 0, \tag{12}$$

where the superscript (k) refers to the current discrete mode. Second time derivatives of the differential variables are computed from first derivatives of the model equations. Partial derivatives of g^* with respect

to the parameters are computed by calling `senrhs` (see below and [11]). These computed values and time will be returned to the user along with an indication of which state condition has changed. The user must then switch the model to the new discrete mode.

When performing a hybrid discrete/continuous simulation and parametric sensitivity analysis a set of transition functions of the form:

$$T_{k+1}^{(k)}(\dot{x}^{(k)}, x^{(k)}, y^{(k)}, u^{(k)}, \dot{x}^{(k+1)}, x^{(k+1)}, y^{(k+1)}, u^{(k+1)}, t, p) = 0 \tag{13}$$

which define the mapping between the final values of the variables in the current discrete mode (k) to the initial values of the variables in the new discrete mode ($k+1$) (initial conditions are a special case of the transition functions) [6]. After DSL48SE returns with consistent final values in the current mode, the user must compute initial values for $\dot{x}^{(k+1)}$, $x^{(k+1)}$, and $y^{(k+1)}$ in discrete mode $k+1$ using the appropriate transition functions. During the next call to DSL48SE after the event, the jump in sensitivities will be computed. The following system of equations is solved for each parameter $i = 1, \ldots, n_p$ to obtain the initial sensitivities in the new mode [6]:

$$\begin{bmatrix} \frac{\partial T_{k+1}^{(k)}}{\partial \dot{x}^{(k+1)}} & \frac{\partial T_{k+1}^{(k)}}{\partial x^{(k+1)}} & \frac{\partial T_{k+1}^{(k)}}{\partial y^{(k+1)}} \\ \frac{\partial F^{(k+1)}}{\partial \dot{x}^{(k+1)}} & \frac{\partial F^{(k+1)}}{\partial x^{(k+1)}} & \frac{\partial F^{(k+1)}}{\partial y^{(k+1)}} \end{bmatrix} \begin{bmatrix} \frac{\partial \dot{x}^{(k+1)}}{\partial p_i} \\ \frac{\partial x^{(k+1)}}{\partial p_i} \\ \frac{\partial y^{(k+1)}}{\partial p_i} \end{bmatrix} = \tag{14}$$

$$- \begin{bmatrix} \frac{\partial T_{k+1}^{(k)}}{\partial \dot{x}^{(k+1)}} & \frac{\partial T_{k+1}^{(k)}}{\partial x^{(k+1)}} & \frac{\partial T_{k+1}^{(k)}}{\partial y^{(k+1)}} \\ \frac{\partial F^{(k+1)}}{\partial \dot{x}^{(k+1)}} & \frac{\partial F^{(k+1)}}{\partial x^{(k+1)}} & \frac{\partial F^{(k+1)}}{\partial y^{(k+1)}} \end{bmatrix} \begin{bmatrix} \frac{\partial \dot{x}^{(k+1)}}{\partial t} \\ \frac{\partial x^{(k+1)}}{\partial t} \\ \frac{\partial y^{(k+1)}}{\partial t} \end{bmatrix} \frac{\partial t}{\partial p_i}$$

$$- \begin{bmatrix} \frac{\partial T_{k+1}^{(k)}}{\partial \dot{x}^{(k)}} & \frac{\partial T_{k+1}^{(k)}}{\partial x^{(k)}} & \frac{\partial T_{k+1}^{(k)}}{\partial y^{(k)}} & \frac{\partial T_{k+1}^{(k)}}{\partial u^{(k)}} & \frac{\partial T_{k+1}^{(k)}}{\partial u^{(k+1)}} & \frac{\partial T_{k+1}^{(k)}}{\partial p_i} & \frac{\partial T_{k+1}^{(k)}}{\partial t} \\ 0 & 0 & 0 & 0 & \frac{\partial F^{(k+1)}}{\partial u^{(k+1)}} & \frac{\partial F^{(k+1)}}{\partial p_i} & \frac{\partial F^{(k+1)}}{\partial t} \end{bmatrix} \begin{bmatrix} \frac{\partial \dot{x}^{(k)}}{\partial p_i} + \frac{\partial \dot{x}^{(k)}}{\partial t}\frac{\partial t}{\partial p_i} \\ \frac{\partial x^{(k)}}{\partial p_i} + \frac{\partial x^{(k)}}{\partial t}\frac{\partial t}{\partial p_i} \\ \frac{\partial y^{(k)}}{\partial p_i} + \frac{\partial y^{(k)}}{\partial t}\frac{\partial t}{\partial p_i} \\ \frac{\partial u^{(k)}}{\partial p_i} + \frac{\partial u^{(k)}}{\partial t}\frac{\partial t}{\partial p_i} \\ \frac{\partial u^{(k+1)}}{\partial p_i} + \frac{\partial u^{(k+1)}}{\partial t}\frac{\partial t}{\partial p_i} \\ I \\ \frac{\partial t}{\partial p_i} \end{bmatrix}.$$

The only additional information that must be provided by the user for this calculation is a subroutine that computes the partial derivatives of the transition conditions. This is described in more detail below. The following section describes the information that must be provided by the user in order to use DSL48SE.

3 Using DSL48SE

The interface to DSL48SE is shown below.

```
      subroutine dsl48se(res, senrhs, neq, t, z, zprime, tout, info,
     1   rtol, atol, idid, rwork, lrw, iwork, liw, rpar, ipar, jac,
     2   nejac, jrow, jcol, jydot, njydot, jdtype, bound,
c.....the following arguments are for state event location
     3   ievent,idiscon,nc,nd,isc,tevt,lievwk,ievwk,lrevwk,revwk,
```

```
     4   stchange,evtderiv,trderivs,rwjump,iwjump)
c### 990804 j.e.tolsma copyright mit
c### dsl48se - dsl48s with state event location and hybrid 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 .ne. 0 then idiscon is the discontinuity
c###                 function in 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### (*) trderivs  - user supplied subroutine (see below)
c### (*) rwjump(np+2*neq)
c###                - real workspace used to hold information for computing
c###                 the sensitivity jump after an event has been located.
c###                 np is the number of parameters of interest, i.e.,
c###                 number of parameters for which sensitivities are
c###                 being computed.
c### (*) iwjump(2+nx)
c###                - integer workspace used to hold information for computing
c###                 the sensitivity jump after an event has been located. nx
c###                 is the maximum number of differential variables in any of
```

```
c###                 the discrete modes.
c###
c###  user-supplied subroutines
c###  ========================
c###  DSL48SE requires information in addition to that required by DSL48S
c###  in order to perform hybrid discrete/continuous dynamic simulation and
c###  parametric sensitivity analysis.  this information is provided through
c###  callback subroutines appearing in the argument list (similar to the
c###  callbacks RES, JAC, and SENRHS used by DSL48S).  the additional callbacks
c###  required by DSL48SE are:
c###     stchange
c###     evtderiv
c###     trderivs
c###  ============================================================================
```

The first 25 arguments are the same as DSL48S and are described in the accompanying documentation [11]. The DSL48S documentation should be read before the remainder of this manual. The DSL48S documentation describes sensitivity analysis in detail, this manual simply describes the additional steps that must be taken in order to perform a hybrid sensitivity analysis. One additional entry in the information array `info` has been added. Entry `info(25)` should be set by the user to zero if the transition functions are simply continuity of the differential variables across the event (i.e., $x^{(k)} - x^{(k+1)} = 0 \quad \forall k$) and set to a nonzero value otherwise. If `info(25)` is nonzero then the user must provide partial derivatives of the transition functions through the callback `trderivs` described below.

The residual, Jacobian, and sensitivity right-hand-side subroutines, `res`, `jac`, and `senrhs`, must perform discontinuity-locked evaluations of the augmented model. The original hybrid DAE model equations, f, must be augmented as follows:

$$f(\dot{z}, z, t; p) = 0 \tag{15}$$

$$z_{nn-n_d+i} - g_i(\dot{z}, z, t; p) = 0 \qquad i = 1, \ldots, n_d \tag{16}$$

where $nn = n_x + n_y$, $z = (x, y)$, and $\{g_i\}_{i=1}^{n_d}$ are the discontinuity functions associated with the current set of state conditions. Note that the controls, $u(t)$, have been removed from the argument list. This is possible since they are explicit functions of time. Similar to DSL48S, the state and time derivative vectors passed to DSL48SE are of dimension $neq = nn(n_p + 1)$, where nn is defined above and n_p is the number of parameters of interest. The discontinuity equations (16) 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** appear directly after the original model variables and the discontinuity equations **must** follow the original model equations. The user-supplied subroutine `res` returns the model equations (15-16) *locked* in the current discrete mode. The user-supplied subroutine `jac` returns the Jacobian matrix of the locked model equations with respect to z and $\dot{z}$. The user-supplied subroutine `senrhs` returns the partial derivatives of locked model equations with respect to the parameters of interest, p. See the DSL48S documentation for a description of the interfaces of these routines. These subroutines are identical for DSL48SE except for the fact that the original model equations are augmented with the discontinuity equations and they must evaluate the model locked in the current discrete mode.

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 nx, the maximum number of differential variables in any of the discrete modes (the maximum dimension of $x^{(k)}$'s) plus the maximum number of discontinuity functions in any of the state conditions (i.e., the maximum entry in `isc(1:nc)`), or nn, whichever is larger. The size of the double precision workspace must be greater than or equal to `(q+1)*nd + q + 5*nn + 40` where `q` is the maximum order of the integration (contained in `info(9)`, which has a default value of 5). Arguments `rwjump` and `iwjump` are double precision and integer arrays, respectively, used to store information required to compute the jump in sensitivities after an event has been located. Array `rwjump` must contain at least $n_p + 2neq$ entries, where as described above, n_p is the number of parameters of interest and $neq = nn(n_p + 1)$ is the total number of states (including discontinuity variables) and sensitivities. Array `iwjump` must contain at least $2 + nx$ entries. Table 1 contains a summary of the additional arguments required by DSL48SE.

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 the interface shown below.

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

This subroutine is used by DSL48SE to determine whether or not a given state condition has changed value. The argument `ichange` is an output integer variable that should be set equal to one if state condition `istate` has changed value 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). Arguments `ns`, `nc`, `nd` are the total number of DAE states and discontinuity variables, total number of state conditions, and total number of discontinuity functions, respectively. Arguments `time`, `z(ns)`, `zdot(ns)`, `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,ns,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 (11)) and computation of the jump in sensitivities. 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 (9)); g^* is the `idiscon`-th discontinuity function of the `istate`-th state condition. Input arguments `ns`, `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:

$$d = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial u}\frac{du}{dt} \tag{17}$$

$$\alpha = \frac{\partial g^*}{\partial t} + \left(\frac{\partial g^*}{\partial u}\right)^T \frac{du}{dt} \tag{18}$$

where F is the augmented DAE model (equations (15) through (16)). Note that these are ***not*** the same as the d an α defined in equation (11). 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.

The third user-supplied subroutine in the DSL48SE argument list is `trderivs`. However, this argument is not currently used and thus, the only valid value of `info(25)` is zero. In future releases, DSL48SE will provide the user with the ability to specify transition conditions associated with the transition from one mode to the next. Subroutine `trderivs` will by used by DSL48SE to compute the partial derivatives of the transition conditions required when computing the jump in sensitivities.

4 Generating the Additional Information with DAEPACK

A substantial amount of information is required to perform numerical integration and parametric sensitivity analysis 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 changed value, a subroutine that returns the locked partial derivatives of the augmented model with respect to $\dot{z}$ and z, and a subroutine that returns the locked partial derivatives with respect to time (for vector d and scalar α). The difficulty in generating this additional information may make the use of DSL48SE prohibitively difficult for large, complicated models. This is particularly true when legacy code is used. Fortunately, the symbolic components of DAEPACK [10] 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 2 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 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 DSL48SE. In addition, the DAEPACK distribution contains an template program file that shows how DSL48SE can be used to perform a hybrid sensitivity analysis. This file can be edited for a particular problem or used as a

reference for incorporating DSL48SE into a larger application. This example code is described in more detail in the following section.

5 Quick Start

The DAEPACK distribution contains a number of example directories containing applications illustrating the use of the numeric components for a variety of basic calculations. The code provided which illustrates DSL48SE performs hybrid parametric sensitivity analysis with a small circuit simulation model originally due to Carver [4]. This example model consists of eight DAEs with 14 parameters. The remainder of this section describes how create, execute, and modify this example. This section assumes that the user has installed DAEPACK on a UNIX/Linux machine.

Suppose the DAEPACK distribution is installed in directory `/home/daepack`. This directory contains a number of subdirectories including `Examples` and `Wrappers`. Both of these directories contain subdirectories named `DSL48SE` containing the code relevant to this example. Directory `Wrappers/DSL48SE` contains the wrapper code and DAEPACK code generation specification files required for constructing the generated code described in the previous section. If the example code is being used as a starting point for other problems, examine the Fortran files in this directory to make sure arrays are dimensioned appropriately (in particular, the `rpar` array in `dsl48se_res.f` which holds the parameters of interest for sensitivity analysis). Directory `Examples/DSL48SE` contains three files: `makefile`, `main.f`, and `res0.f`. For those readers not familiar with makefiles, the file `makefile` describes how to build a given application. By typing "make" at the command-line in this directory, file `makefile` is used to construct the DSL48SE example. File `main.f` is the calculation executive where memory is allocated, initial conditions and other information are provided, and DSL48SE is called repeatedly to perform the hybrid sensitivity calculation. File `res0.f` contains the Fortran code for the model of interest.

File `main.f` is shown below. As stated above, this file contains the code which sets up the problem, performs necessary initializations, and repeatedly calls DSL48SE. The code is shown below.

```
c this program demonstrates how DSL48SE may be used to perform numerical
c integration and parametric sensitivity analysis for a system of
c differential/algebraic equations (DAEs) containing discontinuities. the
c example here is a simple system of eight DAEs originally due to Carver
c (1978), however, the example can be modified and used as a starting point
c for other examples.  comments are provided where modifications should be
c made for new problems.
      program dsl48se_main
      implicit none
      integer maxn,maxnz
c.....change 'maxn' and 'maxnz' to values greater than or equal to the actual
c     number of equations in the model and the maximum number of nonzero
c     entries expected in the sparse jacobian matrix, respectively.
      parameter(maxn=5000,maxnz=10000)
      integer nz,irlist(maxnz),jclist(maxnz),jdtype(maxnz)
      integer info(25)
      double precision atol(maxn),rtol(maxn)
      double precision y(maxn),ydot(maxn),t,tout
      double precision ybounds(2*maxn),delta(maxn)
```

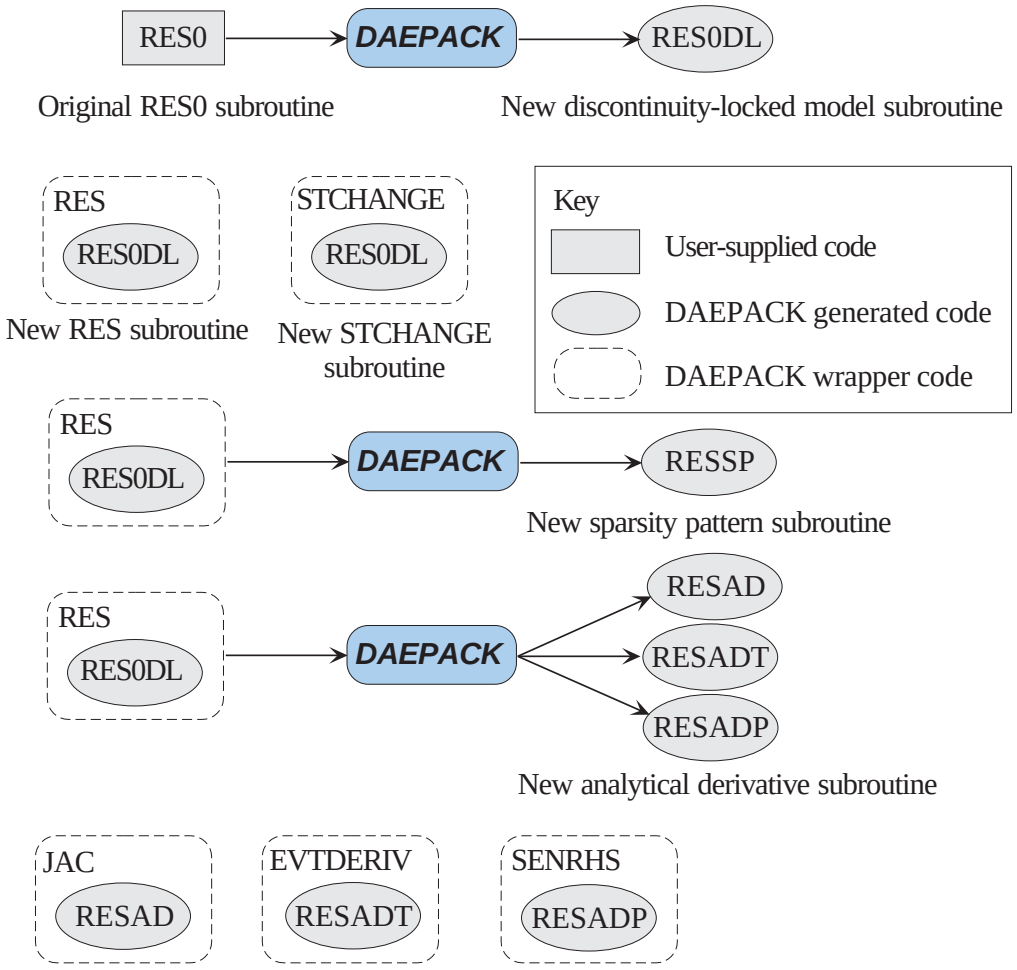

Figure 2: Automatic generation of additional information required for hybrid dynamic simulation and parametric sensitivity analysis with DAEPACK.

```
      integer njydot,jydot(maxn)
      integer idid
c###
      integer lrw,liw
c.....change 'lrw' and 'liw' to values appropriate for the problem.  see
c     DSL48E documentation for appropriate values for these lengths.
      parameter(lrw=60000,liw=10000)
      integer iwork(liw)
      double precision rwork(lrw)
      integer i,j,irow,jcol
      double precision step
      integer ires,ichvar,icode
      integer nsc,ndf
      logical locked
c###
      integer maxsc,maxdf
c.....change 'maxsc' and 'maxdf' to the maximum number of state events and
c     total number of discontinuity functions, respectively.
      parameter(maxsc=100,maxdf=5*maxsc)
      integer isc(maxsc),iscchng(maxsc),ig(maxdf)
      double precision tevt
      integer ievent,idiscon,ny,np,nn,neq,istate
c###
      integer lrpar,lipar
c.....array 'rpar' is used to pass real parameters to the user-supplied
c     subroutines res, jac, and senrhs.  simply make 'lrpar' large enough
c     to hold the desired number of parameters.  array 'ipar' is used both
c     as workspace when using DAEPACK generated discontinuity locking code,
c     to pass the number of real parameters we desire sensitivities with
c     respect to, the indices in 'rpar' corresponding the parameter of
c     interest, and to pass integer parameters to the user-supplied
c     subroutines res, jac, and senrhs.  make 'lipar' greater than
c     3+2*maxsc+maxdf+np plus the number of user-supplied integer
c     parameters.
      parameter(lrpar=1000,lipar=2+2*maxsc+maxdf+1000)
      integer ipar_user(lipar),ipar(lipar)
      double precision rpar(lrpar)
      integer locipar,nipar,nrpar
      integer lievwk,lrevwk
c.....'lievwk' and 'lrevwk' should be set based on the dimension of the
c     current problem.  see the DSL48E documentation to determine how large
c     these values must be.
      parameter(lievwk=2000,lrevwk=5000)
      integer ievwk(lievwk)
      double precision revwk(lrevwk)
      double precision rwjump(5*maxn)
```

```
      integer iwjump(5*maxn)
c###
      double precision alpha

      external res,jac,senrhs,res0,res0ad,res0dl,ressp
      external stchange,evtderiv,trderivs,dsl48se
      external dcopy
c###
c### initialize variable workspace to zero
      do i=1,maxn
         y(i)=0.0d0
         ydot(i)=0.0d0
      end do
c###
c### number of states, parameters, and total number of equations.
      ny=8 ! (CHANGE TO DIMENSION OF YOUR PROBLEM)
      np=1 ! (CHANGE TO NUMBER OF PARAMETERS IN YOUR PROBLEM)
c###
c### integer information array passed to DSL48SE (see documentation)
      info(1)=0  ! integration control, 0=start integration
      info(2)=1  ! tolerance specification, 0=rtol and atol are scalars
      info(3)=1  ! output mode, 0=output only at tout, 1=output every step
      info(4)=0  ! 0=possible to integrate past tout then interpolate
      info(5)=1  ! derivative option, 1=analytical derivative provided
      info(6)=0  ! currently not used
      info(7)=0  ! maximum stepsize, 0=no additional restrictions on stepsize
      info(8)=0  ! initial stepsize, 0=code computes initial stepsize
      info(9)=0  ! integration order, 0=default maximum integration order of 5
      info(10)=0 ! nonnegative solutions, 0=no non negativity constraints
      info(11)=0 ! currently not used
      info(12)=0 ! currently not used
      info(13)=0 ! high index pivoting, 0=don't check for high index pivoting
      info(14)=0 ! output parameter if high index pivoting is necessary
      info(15)=0 ! bounds checking, 0=no dynamic bounds checking
      info(16)=1 ! scaling option, 0=scale iteration matrix
c### parametric sensitivity analysis options
      info(17)=np ! number of parameters for sensitivity analysis
      info(18)=0  ! finite difference option for sensitivity
      info(19)=0  ! error control, 0=states & sensitivities under error control
      info(20)=0  ! zero sensitivities, 0=no sensitivities are identically zero
      info(21)=0  ! perturbation factor for finite difference sensitivities
      info(22)=0  ! ipar marker for location of parameters in rpar
      info(23)=0  ! sensitivity method employed
      info(24)=1  ! initialization of sensitivities (1=yes).
      info(25)=0  ! event transition (0=continuity of differential variables)
c###
```

```
c### specify integer and real parameters in ipar and rpar
c###
c### NOTE: ipar is also used to hold discontinuity locking information.
c### the initial part of ipar holds discontinuity locking information
c### and the remainder holds the user specified integer parameters.
c### specify only the integer parameters in ipar_user - the integer
c### array ipar will be filled automatically to hold the discontinuity
c### locking information and the integer parameters.
      nipar=1+np
      ipar_user(1)=np ! number of parameter for which we require sensitivities
      ipar_user(2)=5  ! indices in 'rpar' of the parameters of interest.

      nrpar=14
      rpar(1)=12.0d0
      rpar(2)=0.24d0
      rpar(3)=12.0d0
      rpar(4)=0.24d0
      rpar(5)=13.64d0
      rpar(6)=-11.64d0
      rpar(7)=-50.0d0
      rpar(8)=0.0d0
      rpar(9)=-11.64d0
      rpar(10)=13.64d0
      rpar(11)=0.0d0
      rpar(12)=-50.0d0
      rpar(13)=10.0d0
      rpar(14)=0.20d0
c###
c### specify consistent initial conditions
c### (CHANGE TO AN APPROPRIATE INITIAL CONDITIONS FOR YOUR PROBLEM).
      y(1)=0.0d0
      y(2)=0.0d0
      y(3)=0.0d0
      y(4)=100.0d0
      y(5)=-100.0d0
      y(6)=83.3333d0
      y(7)=16.6667d0
      y(8)=-183.3333d0
      ydot(1)=416.6667d0
      ydot(2)=0.0d0
      ydot(3)=0.0d0
      ydot(4)=0.0d0
      ydot(5)=0.0d0
      ydot(6)=0.0d0
      ydot(7)=0.0d0
      ydot(8)=0.0d0
```

```
c###
c### open data file
      open(unit=1,file='dsl48se.states')
      open(unit=2,file='dsl48se.sensitivities')
c###
c### specify integration interval
      t=0.0d0
      tout=0.05d0
c###
c### main loop for overall integration (over all subdomains)
200   continue
c###
c### lock residual in current model
      locked=.false.
      istate=0
      call res0dl(ny,t,y,ydot,rwork,ires,ichvar,rpar,ipar_user,
     1   nsc,ndf,isc,iscchng,rwork(ny+1),locked,istate,ig)
c###
c### fill integer parameter array ipar so that initial part holds the
c### discontinuity locking information and the remainder holds the
c### parameters.
      ipar(1)=nsc
      ipar(2)=ndf
      do i=1,nsc
         ipar(2+i)=isc(i)
      end do
      locipar=3+2*nsc+ndf
      do i=1,nipar
         ipar(locipar+i-1)=ipar_user(i)
      end do
c###
c### initialize discontinuity variables
      do i=1,ndf
         y(ny+i)=rwork(ny+i)
      end do
c###
c### add number of discontinuity equations to original equations and multiply
c### by (np+1) to get dimension of overall system of equations.
      neq=(ny+ndf)*(np+1)
c###
c### get occurrence information
      call ressp(ny+ndf,t,y,ydot,rwork,ires,ichvar,rpar,ipar,
     1   nz,irlist,jclist,iwork(16))
c###
c### adjust indices of jclist corresponding to ydot (i.e., make them
c### the same as y) and fill jydot and jdtype. assume all derivatives
```

```
c### are analytical (i.e., no constant derivative entries).
      njydot=0
      nn=ny+ndf
      do i=1,nz
         if(jclist(i).gt.nn) then
c###        this is a derivative w.r.t. a time derivative
            jdtype(i)=-2
            jclist(i)=jclist(i)-nn
            njydot=njydot+1
            jydot(njydot)=i
         else
c###        this is a derivative w.r.t. a state variable.
            jdtype(i)=1
         end if
      end do
c###
c### specify tolerances
      do i=1,neq
         atol(i)=1.0d-8
         rtol(i)=1.0d-8
      end do
c###
c### display information about current mode
      write(*,*) 'Current mode information'
      write(*,5000) nsc,ndf
      if(nsc.lt.20) write(*,5001) (isc(i),i=1,nsc)
      if(ny.lt.30)  write(*,5002) (y(i),i=1,ny)
      if(ndf.lt.30) write(*,5003) (y(ny+i),i=1,ndf)
c###
c### initialize df/dt if this is the first call to dsl48se
      if(t.eq.0.0d0) then
         istate=0
         idiscon=0
         call evtderiv(icode,istate,idiscon,neq,nsc,ndf,isc,
     1      t,y,ydot,rpar,ipar,rwork(41),alpha)
      end if
c###
c### beginning of subdomain integration loop
c### step is used for displaying integration step to screen
      step=0.0d0
100   continue
c###
c### display integration interval to screen
      write(*,2000) t,t+step
2000  format('.. integrating from ',d10.4,' to ',d10.4)
c### call dsl48se to take integration step
```

```
      call dsl48se(res,senrhs,neq,t,y,ydot,tout,info,rtol,atol,
     1   idid,rwork,lrw,iwork,liw,rpar,ipar,jac,nz,
     2   irlist,jclist,jydot,njydot,jdtype,ybounds,
     3   ievent,idiscon,nsc,ndf,isc,tevt,lievwk,ievwk,
     4   lrevwk,revwk,stchange,evtderiv,trderivs,rwjump,iwjump)
c### write DAE states to data file for later visualization
      write(1,2001) t,(y(i),i=1,ny)
      write(2,2001) t,(y(i),i=ny+ndf+1,neq-ndf)
2001  format(e10.4,500(' ',e14.6))
c###
c### check for event
      if(ievent.ne.0) then
         write(*,*) '.. Event identified at time: ',tevt
         write(*,*) '   Event = ',ievent,', discontinuity function = ',
     1      idiscon
c###
c### re-initialize system

c### set current time to state event time
         t=tevt

c### compute consistent states in new mode
         call reinit(icode,ny,t,y,ydot,nz,irlist,jclist,rwork,lrw,
     1      iwork(16),liw-16,res0,res0ad,rpar,ipar(locipar))
         if(icode.ne.0) goto 130
c###
c### begin integration in new mode.
         write(*,*) 'Current time (after reinitialization) is ',t
         info(1)=-1
         step=0.0d0
         goto 200

      end if

      step=rwork(3)

      if(idid.eq.1)              goto 100
      if(idid.eq.2.or.idid.eq.3) goto 110
c###
c### failure in dsl48se/dsl48s - exit with error message issued
      goto 120
c###
c### successful completion
110   write(*,2002) idid
2002  format(/,'.. integration completed successfully, idid=',i4)
      goto 4000
```

```
c### error return from dsl48s - report
120   write(*,2003) idid
2003  format(/,'!! error return from dsl48s/dsl48se, idid=',i4)
      goto 4000
c### error return from dsl48s - report
130   write(*,2004) icode
2004  format(/,'!! error return from reinit, icode=',i4)
      goto 4000
3000  format('!! return code from ',a6,': ',i2)
3001  format('!! error return')
c###
c### done - clean-up, display statistics, and exit
4000  continue
c###
5000  format('There are ',i3,' state conditions containing ',
     $   i3,' discontinutiy functions')
5001  format('Number of discontinuity functions per state condition',
     $   /,100(2x,i1))
5002  format('DAE states (initial values)',/,
     $   100(2x,d10.4))
5003  format('Discontinuity functions (initial values)',/,
     $   100(2x,d10.4))

      write(*,6000)
      write(*,6001) iwork(11)
      write(*,6002) iwork(12)
      write(*,6003) iwork(13)
      write(*,6004) iwork(14)
      write(*,6005) iwork(15)

6000  format(/,'Integration Statistics',/,'~~~~~~~~~~~~~~~~~~~~~~')
6001  format('              Number of steps taken: ',i7)
6002  format('     Number of residual evaluations: ',i7)
6003  format('     Number of Jacobian evaluations: ',i7)
6004  format('      Number of error test failures: ',i7)
6005  format('Number of convergence test failures: ',i7)

      close(1)
      close(2)
      end
c###
      subroutine reinit(icode,n,t,y,ydot,nz,irlist,jclist,rwork,
     1   lrw,iwork,liw,res0,res0ad,rpar,ipar)
      implicit none
      integer icode,n,nz,irlist(nz),jclist(nz)
      integer lrw,liw,iwork(liw),ipar(*)
```

```
      double precision t,y(n),ydot(n),rwork(lrw),rpar(*)
      external res0,res0ad

      integer i,k,index,iter,maxiter
      parameter(maxiter=10)
      integer ires,ichvar,jdtype
      double precision eps
      parameter(eps=1.0d-8)
      double precision dznorm,dnrm2
      external dnrm2

      integer ne
      integer maxn,maxne
      parameter(maxn=500,maxne=10000)
      integer irn(maxne),jcn(maxne)
      double precision ajac(maxne),delta(maxn),dz(maxn)

      integer nx,idiffmap(maxn)

      integer job,la
      integer keep(1000),info(12),icntl(9)
      double precision cntl(5)
      double precision rinfo,error(3)
      logical trans

      logical isdifferential
      external isdifferential

      external ma48id,ma48ad,ma48bd,ma48cd
c###
c### initialize ma48 control parameter arrays
      call ma48id(cntl,icntl)

      do iter=1,maxiter
c###
c### compute residuals and Jacobian matrix
         call res0ad(n,t,y,ydot,delta,ires,ichvar,rpar,ipar,
     1      ajac,nz,irlist,jclist,iwork)
c###
c### determine current set of differential variables.
         do i=1,n
            idiffmap(i)=0
         end do
         do i=1,nz
            index=jclist(i)-n
            if(index.gt.0) then
```

```
               idiffmap(index)=1
            end if
         end do
         nx=0
         do i=1,n
            if(idiffmap(i).ne.0) then
               nx=nx+1
               idiffmap(nx)=i
            end if
         end do
c###
c### modify Jacobian matrix by removing entries corresponding to
c### partial derivatives with respect to differential variables.
         k=0
         do i=1,nz
            if(jclist(i).gt.n) then
            ! this is a partial derivative w.r.t. a time derivative.
               k=k+1
               ajac(k)=ajac(i)
               irn(k)=irlist(i)
               jcn(k)=jclist(i)-n ! map variable number to [1,n]
            else if(.not.isdifferential(jclist(i),nx,idiffmap)) then
            ! this is a partial derivative w.r.t. an algebraic variable.
               k=k+1
               ajac(k)=ajac(i)
               irn(k)=irlist(i)
               jcn(k)=jclist(i)
            end if
         end do
         ne=k       ! set new number of nonzero entries in Jacobian
         la=3*ne    ! compute factorization workspace.
c###
c### factor jacobian matrix
C#########################################################################
C#################### dense matrix option LINPACK ########################
C#########################################################################
C          call sptodn(icode,n,ne,len,ajac,irn,jcn,n,rwork)
C          if(icode.ne.0) then
C             write(*,*) 'Insufficient space in sptodn to uncompress ',
C      1         'the sparse matrix'
C             return
C          end if
C          call dgefa(ajac,n,n,keep,icode)
C          if(icode.ne.0) then
C             write(*,*) 'Singular matrix in dgefa'
C             return
```

```
C          end if
C#########################################################################
C#################### sparse matrix option HARWELL ######################
C#########################################################################
         job=1
         call ma48ad(n,n,ne,job,la,ajac,irn,jcn,keep,cntl,
     1      icntl,iwork,info,rinfo)
         if(info(1).ne.0) then
            write(*,3000) 'ma48ad',info(1)
            if(info(1).lt.0) then
               write(*,3001)
               icode=info(1)
               return
            end if
         end if
         call ma48bd(n,n,ne,job,la,ajac,irn,jcn,keep,cntl,
     1      icntl,rwork,iwork,info,rinfo)
         if(info(1).ne.0) then
            write(*,3000) 'ma48bd',info(1)
            if(info(1).lt.0) then
               write(*,3001)
               icode=info(1)
               return
            end if
         end if
c#########################################################################
c###
c### perform backsubstitution
c#########################################################################
c#################### dense matrix option LINPACK #######################
c#########################################################################
C       job=0
C       call dcopy(n,delta,1,dz,1)
C       call dgesl(ajac,n,n,keep,dz,job)
C#########################################################################
C#################### sparse matrix option HARWELL ######################
C#########################################################################
         trans=.false.
         call ma48cd(n,n,trans,job,la,ajac,irn,keep,cntl,
     $      icntl,delta,dz,error,rwork,iwork,info)
         if(info(1).ne.0) then
            write(*,3000) 'ma48cd',info(1)
            if(info(1).lt.0) then
               write(*,3001)
               icode=info(1)
               return
```

```
           end if
         end if
c#######################################################################
c###
c### update iterates
         do i=1,n
            if(isdifferential(i,nx,idiffmap)) then
            ! this is a differential variable - update the time derivative.
               ydot(i)=ydot(i)-dz(i)
            else
            ! this is an algeraic variable
               y(i)=y(i)-dz(i)
            end if
         end do
c###
c### compute norm of step
         dznorm=dnrm2(n,dz,1)
c###
c### display iteration number and norm
         write(*,2003) iter,dznorm

         if(dznorm.lt.eps) then
            icode=0
            write(*,2004)
c            write(*,2005) (y(i),i=1,n)
c            write(*,2006) (ydot(i),i=1,n)
            goto 4000
         end if

      end do
c###
c### unable to converge
      icode=-1
      write(*,2007)
      goto 4000
c###
2000  format('Reinitialization calculation')
2001  format('- there are ,',i2,' differetial variables ',
     $   'with indices: ',100(i2,2x))
2002  format('- values will be fixed at: ',
     $   100(d10.4,2x))
2003  format('.. iteration ',i2,', ||dx|| = ',d10.4)
2004  format('.. iteration converged')
2005  format('.. states',/,5(d10.4,2x))
2006  format('.. time derivatives',/,5(d10.4,2x))
2007  format('!! iteration failed to converge')
```

```
2008  format('.. step',/,5(d10.4,2x))
3000  format('!! return code from ',a6,': ',i2)
3001  format('!! error return')
c
4000  continue
      return
      end
c###
      subroutine trderivs()
      implicit none
      write(*,*) 'INSIDE TRDERIVS - CURRENTLY A DUMMY ARGUMENT'
      return
      end
```

The variables declared in the code shown above are described elsewhere in this document and in the DSL48S manual [11]. As described above, this code may be used as a starting point for other problems. Comments are provided where modification may need to be performed when solving other problems. The beginning of this code contains the variable declarations for the variables and parameters used by DSL48SE. When solving other problems, look at the parameter statements in the code shown above and make sure they are suitable. A number of external subroutines are declared at the beginning of this code. Most of these are constructed automatically by DAEPACK when "make" is typed at the command line. Figure 2 contains a description of the files that are constructed automatically.

A possible point of confusion are the arrays `rpar` and `ipar` passed to the callback functions in the DSL48SE subroutine argument list. Double precision array `rpar` contains the values of the parameters passed to the model. Sensitivity analysis may be performed with respect to one or more of these parameters using information in `ipar` described next. Integer array `ipar` is used to pass integer information to the model. The discontinuity-locked model constructed automatically by DAEPACK requires additional information passed to it that is not in the argument list of the residual subroutine. This is partly due to the fact that the user may choose not to use DAEPACK to construct the discontinuity-locked model, so arguments specific to the DAEPACK generated code are not placed in the argument list. (See Appendix A for a description of the discontinuity-locked model.) Consequently, some of the entries of `ipar` are used to hold this information so that it is available within the generated discontinuity-locked residual routine. Table 4 describes the entries held in the `ipar` array. The first `2+2*nsc+ndf` entries of `ipar` contain information used during the discontinuity-locked evaluation of the model. Since the number of state conditions and discontinuity functions, `nsc` and `ndf`, respectively, may potentially change from one discrete mode to the next, a second integer parameter array, `ipar_user`, should be filled with the additional information described in Table 4 that do not change from one discrete mode to the next. The first entry of `ipar_user` must contain the number of parameters to perform sensitivity analysis with respect to, `np`, the next `np` entries must contain the indicies in `rpar` corresponding to the parameters of interest, and the remaining entries can contain any integer values that need to be passed to the model. Once `nsc` and `ndf` are determined for the current discrete mode, the contents of `ipar_user` are appended to the end of `ipar`.

After the program variables are declared, the number of DAE states (not including discontinuity variables) and the number of parameters for which we desire sensitivities with respect to are set (see comments in code). This is followed by assigning appropriate values to the `info` array. Comments are provided in the code that indicate what options the elements of `info` correspond to and more information

can be found in the DSL48S documentation. Next, the user-specified integer and real parameters are set and initial conditions specified.

The next item of interest in the code above is the call to the discontinuity-locked model, `res0dl`, with argument `locked` set to `.false.`. This subroutine is constructed automatically by DAEPACK. Calling this code with argument `locked` equal to `.false.` determines the currently active mode (i.e., identifies which clauses in IF statements are currently active). All subsequent calls to this model with `locked` equal to `.true.` will evaluate the same model, regardless of the current values of the logical expression of the IF statements. Also returned from subroutine `res0dl` is the values of the discontinuity functions and information about the current discrete mode: total number of state conditions, `nsc`, total number of discontinuity functions, `ndf`, and number of discontinuity functions in each state condition, `isc(1:nsc)`. As described above, this information is stored in `ipar`.

After the model has been locked in the current discrete mode, the sparsity pattern is determined by calling the DAEPACK generated code `ressp`. When constructing the sparsity pattern, the state variables are indexed from one to `ny+ndf` and the time derivatives are indexed from `ny+ndf+1` to `2*(ny+ndf)`. However, DSL48SE requires the time derivatives to have the same indices as their corresponding state variables (see the DSL48S documentation). Consequently, the code following the sparsity pattern evaluation, adjusts the indices of the sparsity pattern corresponding to time derivatives and initializes the arrays `jdtype` and `jydot` (once again, see the DSL48S documentation for a description of these arrays).

After setting the integration tolerances and displaying some information, the partial derivatives of the model with respect to time are evaluated and stored in `rwork(41:41+ny+ndf+1)`. This is only performed at the beginning of the integration and required when using DSL48SE to evaluate the initial sensitivities. This feature is actually performed by DSL48S and is described in the associated documentation.

At this point, DSL48SE is called repeatedly to advance the numerical integration and parametric sensitivity analysis. After each time step (i.e., call to subroutine `dsl48se`), the value of argument `ievent` is checked. If `ievent` is zero then no event has occurred and the calculation is advanced by calling `dsl48se` again. If `ievent` is nonzero then an event has occurred. The value of `ievent` is the index of the state condition that has changed value and the value of argument `idiscon` is the index of the discontinuity function in this state condition that actually triggered the value to change. Furthermore, arguments `tevt`, `y`, and `ydot` contain the polished values of the event time, DAE states and time derivative, respectively. At this point, subroutine `reinit` is called to reinitialize the states and time derivatives for the new discrete mode. If the event hasn't occurred then the return code `idid` is checked to determine if the integration has completed successfully or terminated with errors (`ievent` will be zero if `idid` indicates an error has occurred).

Subroutine `reinit` performs the reinitialization calculation at the event using the Newton-Raphson method. This subroutine assumes that the differential variables remain continuous across the event (i.e., $x^{(k)} = x^{(k+1)}$, using the notation in DAE (2) where the superscript denotes the discrete mode). Two subroutines are passed as arguments to this subroutine, `res0`, the original user-supplied residual evaluator (not the discontinuity-locked model constructed by DAEPACK) and `res0ad`, the subroutine constructed automatically by DAEPACK for evaluating the partial derivatives of the original model with respect to the DAE states and time derivatives. Subroutine `res0ad` performs three tasks: 1) evaluates the sparsity pattern of the matrix, 2) computes the nonzero entries of the matrix, and 3) computes the residuals of the original model. Subroutine `reinit` provides two options for the factorization of the iteration matrix: sparse linear algebra using the Harwell MA48 routines and dense linear algebra using Linpack. The dense option is commented out in the code above. Both linear algebra options use the sparse matrix returned from the DAEPACK generated derivative code `res0ad`. If dense linear algebra is used then the sparse matrix is converted to dense storage using the DAEPACK utility routine `sptodn`. After entering the main

iteration loop, the iteration matrix is evaluated. Using the sparsity pattern returned from `res0ad`, the differential variables are identified, storing the information in `nx` and `indxx(1:nx)`. Next, the iteration matrix is modified so that the differential variables identified in the previous step are held constant during this iteration process. During this matrix updating, the new number of nonzero entries are determined and assigned to `ne` and the workspace for the factorization used by MA48 is stored in `la`. In the code above, this workspace size is set equal to three times the number of nonzero entries. This factor may need to be changed for other problems (see Harwell documentation). If dense linear algebra is used, then `la` must simply be set equal to the number of entries in the dense iteration matrix (`n*n`). Once the modified matrix is constructed the variables updates are computed using a standard Newton-Raphson step. This process is repeated until convergence, an error occurs, or the number of iterations exceeds some maximum value (set in the code). If the iteration was successful, `reinit` returns with the states initialized in the new discrete mode. During the next call to `dsl48se`, the jump in sensitivites will be computed and the calculation will begin in the next subdomain.

The code shown below is the original residual file provided with the DAEPACK distribution, `res0.f`.

```
subroutine res0(neq,t,y,ydot,delta,ires,ichvar,rpar,ipar)
implicit none
integer neq,ires,ichvar,ipar(1)
double precision t,y(neq),ydot(neq),delta(neq),rpar(14)
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
double precision r3,l3

a1=rpar(1)
a2=rpar(2)
a3=rpar(3)
a4=rpar(4)
a5=rpar(5)
a6=rpar(6)
a7=rpar(7)
a8=rpar(8)
a9=rpar(9)
a10=rpar(10)
a11=rpar(11)
a12=rpar(12)
r3=rpar(13)
l3=rpar(14)

x1dot=ydot(1)
x2dot=ydot(2)
x1=y(1)
x2=y(2)
x3=y(3)
v1=y(4)
v2=y(5)
v3=y(6)
```

```
      z3=y(7)
      z4=y(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.
     1   (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.
     1      (x1.lt.0.0d0.and.v1.lt.v3)) then
            delta(7)=x1dot-0.0d0
            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
```

This small example is adapted from Carver [4] and represents an electrical circuit model. The values of the constants and the initial values for integration can be found in the two codes shown above. With these two codes (`res0.f` and `main.f`), the makefile, and the DAEPACK distribution, all the user must do is type "make" in the directory containing the source. All of the additional code required when using DSL48SE is constructed automatically with DAEPACK and an application named `dsl48se` is created. Running this code produces two data files, one containing the state variable trajectors, some of which are shown in Figure 3, and the other containing the hybrid sensitivity trajectories, some of which are shown in Figure 4. Although small, this example illustrates the power of handling discontinuities properly during numerical integration and sensitivity analysis. Unlike the previous example shown in this document where the sensitivity trajectory is incorrect, with this example, the integration fails at the second event and the sensitivities that are computed up to this point are incorrect.

6 Distribution

DAEPACK component DSL48SE is currently available within the DAEPACK libraries

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

and

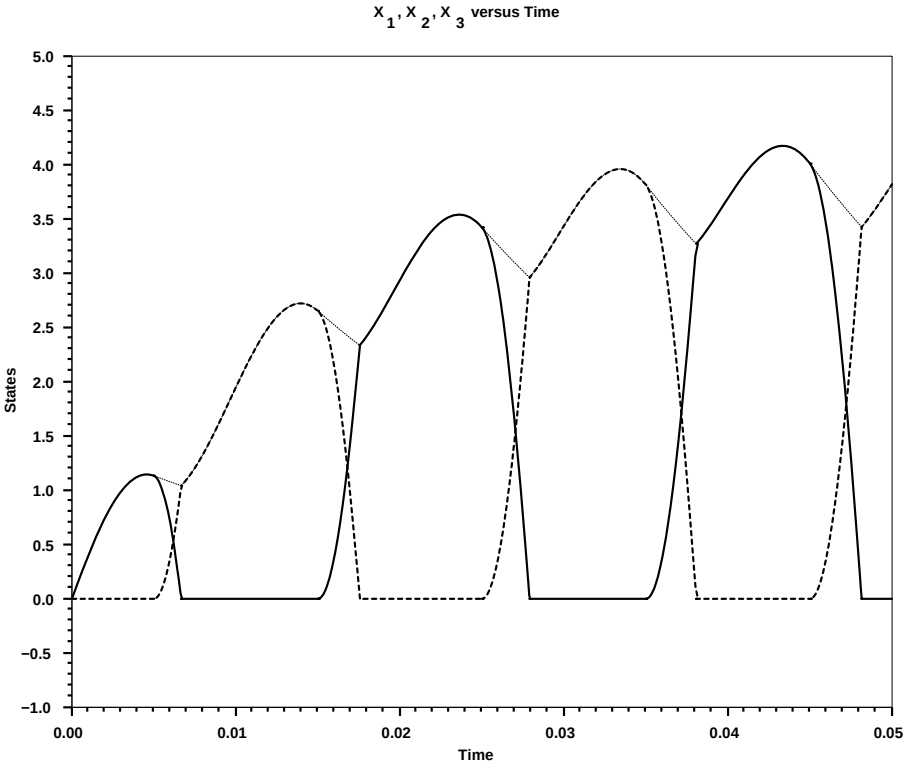

Figure 3: State trajectories for Carver example. Solid line is x_1, dashed line is x_2, and thin dotted line is x_3.

`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 DSL48SE. A complete description of which Harwell routines is given in [13]. 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.

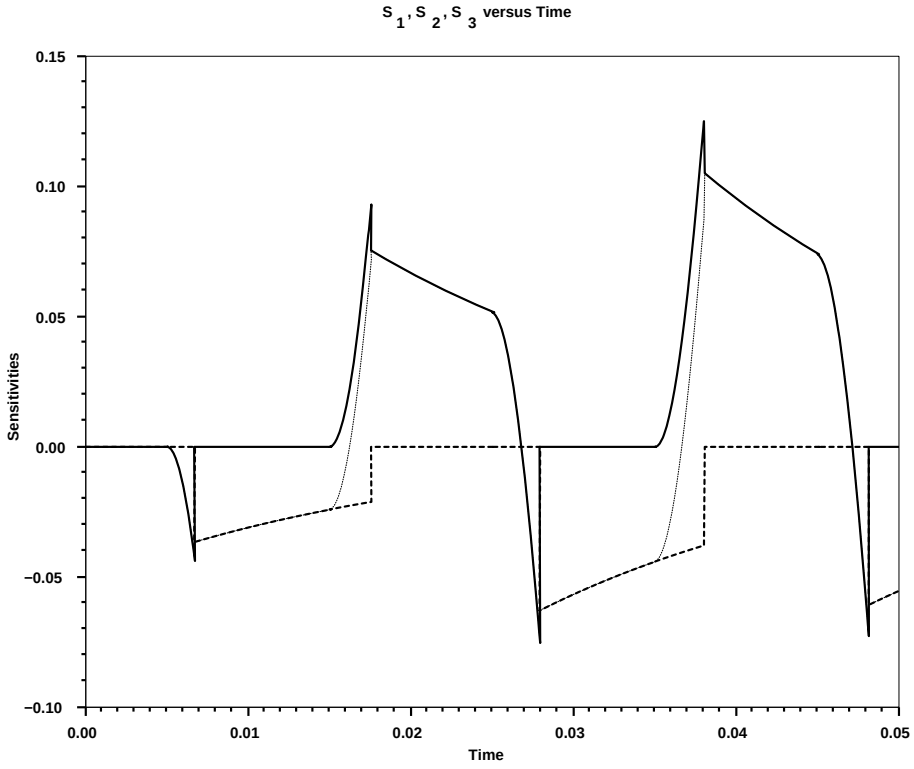

Figure 4: Sensitivity trajectories ($s_1 = \partial x_1/\partial a_5$, $s_2 = \partial x_2/\partial a_5$, and $s_3 = \partial x_3/\partial a_5$) for Carver example. Solid line is s_1, dashed line is s_2, and thin dotted line is s_3.

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] M. B. Carver, *Efficient integration over discontinuities in ordinary differential equation simulations*, Mathematics and Computers in Simulation, XX (1978), pp. 190–196.

[5] W. F. Feehery, J. E. Tolsma, and P. I. Barton, *Efficient sensitivity analysis of large–scale differential–algebraic systems*, Applied Numerical Mathematics, 25 (1997), pp. 41–54.

[6] S. Galán, W. F. Feehery, and P. I. Barton, *Parametric sensitivity functions for hybrid discrete/continuous systems*, Applied Numerical Mathematics, 31 (1999), pp. 17–47.

[7] T. Maly and L. R. Petzold, *Numerical methods and software for sensitivity analysis of differential–algebraic systems*, Applied Numerical Mathematics, 20 (1996), pp. 57–79.

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

[9] 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.

[10] 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.

[11] ——, *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.

[12] ——, *Numerical integration with robust state event location. DSL48E manual*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.

[13] ——, *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.

[14] J. E. Tolsma and P. I. Barton, *Hidden discontinuities and parametric sensitivity calculations.* submitted, SIAM Journal on Scientific Computing, 2000.

Table 1: Summary of additional arguments required by DSL48SE. All other arguments are the same as DSL48S (except that the residual, Jacobian, and sensitivity right-hand-side 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.
input	subroutine	`trderivs`	User supplied subroutine returning partial derivatives of transition functions.
—	double	`rwjump(*)`	Real workspace used to compute sensitivity jump.
—	integer	`iwjump(*)`	Integer workspace used to compute sensitivity jump.

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	`ns`	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(ns)`	Current values of DAE states and discontinuity variables.
input	double	`zdot(ns)`	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.

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

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	`ns`	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(ns)`	Current values of DAE states the derivatives are to be evaluated at.
input	double	`zdot(ns)`	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(s)`	Computed partial derivatives (see equation (17))
output	double	`alpha`	Computed partial derivatives (see equation (18))

Table 4: Summary of the information stored in `ipar`.

`ipar(1)`	`nsc`, the total number of state conditions in the current mode.
`ipar(2)`	`ndf`, the total number of discontinuity functions in the current mode.
`ipar(2+1:2+nsc)`	`isc(1:nsc)`, the number of discontinuity functions in each state condition.
`ipar(2+nsc+1:2+2*nsc+ndf)`	workspace used during a locked model evaluation.
`ipar(2+2*nsc+ndf+1)`	`np`, the number of parameters for sensitivity analysis.
`ipar(2+2*nsc+ndf+2: 2+2*nsc+ndf+np+1)`	the indices in `rpar` corresponding to the parameters of interest.
`ipar(2+2*nsc+ndf+np+2:*)`	all remaining entries may be used by the user to pass in model specific integer parameters.