Tools for Modeling and Simulation of Hybrid Systems – A Tutorial Guide

James H. Taylor
Department of Electrical & Computer Engineering
University of New Brunswick
Fredericton, NB Canada E3B 5A3
E-mail: *jtaylor@unb.ca*

12 August 1996

Revised: 4 September 1999

Overview: Previous research in the area of modeling and simulation of hybrid systems led to the development of a general hybrid systems modeling language (HSML), a subset of which has been implemented by extending MATLAB. This brief tutorial guide describes the algorithmic implementation of the HSML ideas and language constructs for dealing with state-event handling in continuous-time system components. Specifically, we describe how the standard MATLAB model framework and integration algorithms have been extended to support these phenomena, and present a number of examples that illustrate the approach and demonstrate using the software within the MATLAB environment. Note that the examples discussed here can be inspected and run if you visit our distribution site, http://www.ece.unb.ca/jtaylor/HS_software.html and download *all* the files provided in (HYBRID_SIM.ZIP); only a few results are given here, to keep this guide reasonably short.

The present version of this guide has been revised, to reflect modifications that had to be made to accommodate the syntax changes in MATLAB 5; the software has been tested successfully on MATLAB 5.3, MATLAB 6.1, MATLAB 6.5 (release 13 for unix) and MATLAB 7.4 (release R2007a for unix); if you have difficulty on other installations please let me know.

We have extended this methodology to handle hybrid systems comprised of a mixture of discrete- and continuous-time components. The main focus of that effort was precise timing and coordination of state and time events during simulation. The resulting software has not been tested sufficiently for distribution; it will be made available on http://www.ece.unb.ca/jtaylor/HS_software.html eventually ...

1 Introduction

The HSML language [1, 2] was designed to support a broad definition of a hybrid system, which we may express informally as being an arbitrary interconnection of components that are arbitrary instances of continuous-time, discrete-time and logic-based systems. Requirements for HSML focused particularly on the rigorous characterization and execution of "events", both discrete- and continuous-time, that cause discontinuous changes in system trajectories and/or the model itself. In this respect, there is much commonality between the HSML project and recent developments by Elmqvist, Cellier *et al.* in the area of object-oriented modeling [3]; for a detailed view of state-event handling see especially [4, 5].

This brief tutorial guide outlines the completion of the first phase in implementing a subset of the HSML concept in a working modeling and simulation environment, MATLAB [6]. Preliminary steps are documented in some detail in [7], and a short but more up-to-date description may be found in [8]. Here we again focus narrowly on the issues surrounding state-event handling in continuous-time components (CTCs) and provide several illustrative examples to demonstrate the use and efficacy of the approach.

Here we consider CTCs that may be represented as[1]:

$$\begin{aligned} \dot{x} &= f(x, u, m, t) \\ y &= h(x, u, m, t) \end{aligned} \tag{1}$$

where x is the state vector, y is the output vector, u is a numeric input signal (continuous-time), m is comprised of a finite alphabet of numeric or symbolic input variables that characterizes the "mode" of the model, and t is the time; in general u and m are vectors. Of particular importance to the present exposition, the **mode** input m is included to provide means of controlling the model's structure and coordinating its behavior with the numerical integration process in state-event handling, as described below. Based on the types of physical phenomena of interest, we restrict elements of m to take on the values -1 , 0 , $+1$.

In our implementation, state events are characterized by *zero-crossings,*

$$S(x, m, t) = 0 \tag{2}$$

where S is a general expression involving the state, time and mode of the CTC model. An arbitrary state change in the CTC model can be classified as a **negative-going event** (i.e., one in which S becomes negative), an **on-boundary event** (S becomes and remains equal to zero for a period of time), or a **positive-going event**. Note that this framework provides partial support for models that undergo structural changes

[1]The specific class of CTC that can be modeled depends on the simulator's integration methods; MATLAB cannot handle differential algebraic equations (DAEs), so we have restricted ourselves to ordinary differential equations and simplified the variable types in comparison with [1, 2, 7].

(e.g., changes in the definition or number of state variables) [13]; e.g., in the case of mechanical subsystems engaging, the number of states decreases, which we can model (inefficiently) by carrying along redundant or "dummy" states during the intervals when the state-space dimension is reduced. Finally, we include provision for **instantaneous reset** of the model state variables at an event, according to

$$x^+ = x(t_e^+) = r(x(t_e^-), m, t_e^-) \tag{3}$$

where r is also an arbitrary expression and t_e is the event time. This feature is useful in resetting velocities after engagement to conserve momentum, for example.

Given the above problem definition, the correct handling of state events is as follows:

1. The model should not be allowed to switch during a numerical integration step. Integrate as usual as long as the variable S does not change sign; each integration point is treated as a "trial" point until the sign condition is checked; if no sign change has occurred, the point becomes "accepted".
2. When a sign change is detected, the trial point is discarded and an iterative procedure is initiated (within the simulator) to find the time step h^* such that S has just passed zero, e.g., for a positive-going event $S \in (0, \epsilon)$. The model is not allowed to switch during this procedure.
3. The integrator produces an accepted point on the switching curve (Eqn. 2) and then signals the model to switch (e.g., by changing `mode` from – 1 to 0 or +1 depending on the nature of the event).
4. States are reset, if necessary, and normal integration proceeds from that point.

This process requires coordination between the model and simulation package, as illustrated below using MATLAB extensions.

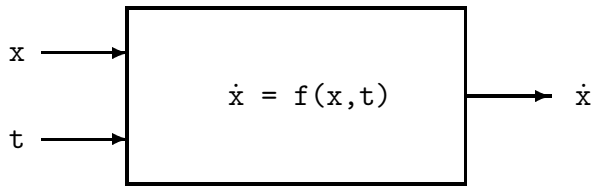
2 Extended Model Schema

One significant extension needed in MATLAB for modeling and simulating state events in CTCs is in the input/output structure of the model. The existing and extended schema are depicted in Fig. 1:

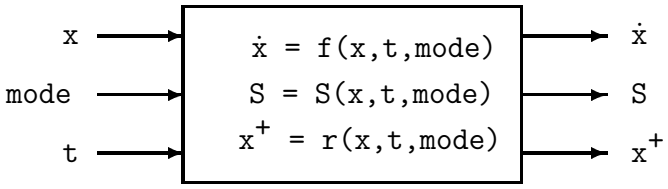
The additional outputs are the **state-event signal** S (Eqn. 2) and the **state-reset vector** x^+ (Eqn. 3); the new input `mode` allows the numerical integration routine to tell the model to switch according to the state event just detected. Again, note that S and m (mode) may be vectors, to support multiple state events and switching boundaries. Two examples are provided below to illustrate these extensions: a relay switching system and an electro-mechanical system with stiction.

2.1 Relay Switching System Model

A simple example of the structure of this extended model is as follows (this is a listing of `relay_seh.m`, see Section 4.1):



(a) Standard MATLAB model schema



(b) Extended MATLAB model schema

Figure 1: MATLAB model input/output structures

```
function [xdot,phi,reset] = relay(t,x,mode)
   % model of the relay switching system, d^2x/dt^2 = - sign(x) --
   % "mode" switches when phi = x(1) passes through zero, and that
   % instantiates the state event (causes the integrator to switch
   % mode from -1 to 1 or vice versa).  JH Taylor, 8 March 1996.
if isempty(mode), % initialization section
   % Set phi (S) to be consistent with the IC, so that mode
   % will be initialized correctly -- e.g., if x(1) = 0,
   % then x(2) governs mode, i.e. x(2) > 0 => mode = +1 etc.
   if x(1) == 0, phi = x(2);
   else phi = x(1);
   end
   xdot = []; reset = []; % to prevent matlab 5 warning msgs!!
   return
end
% Define the mode-dependent model and switching flag:
xdot(1) = x(2);
xdot(2) = -mode;
phi = x(1);
reset = []; % to prevent matlab 5 warning msgs!!
```

The initialization section (`if isempty(mode), ... return, end`) is vital. In it, you must define the initial value of the switching function (S, but called `phi` in this code) to be consistent with initial conditions on the state variables; to be safe you should consider all eventualities (in this example we consider not only which side of the switching boundary we are on if $x_1 \neq 0$ but which side we will move into if we start on the boundary ($x_1 = 0$)). The remaining code contains the *mode-dependent* system differential equations (note that the relay will switch only when mode changes from -1 to +1 or *vice versa*; we do *not* use the relation `xdot(2) = -sign(x(1))`!) and the switching function ($S =$ `phi = x(1)`).

2.2 Model of a System with Stiction

The following more detailed example shows how to exploit the three-level mode option and state reset in modeling an electro-mechanical system with stiction (see Section 4.4):

```
function [xdot,phi,reset] = stiction(t,x,mode)
% electro-mechanical system with stiction and saturation
%
% mode switches from +/-1 to 0 when velocity passes through zero,
% and transitions back to +/-1 when there is sufficient torque to
% overcome stiction.  States are x(1) = position, x(2) = velocity;
% the model is identical to Taylor & Strobel, 1985 ACC paper.
%                                          JH Taylor, 8 June 1996.
%
global v_in omega % parameters needed to define input signal
if isempty(mode), % initialization section
   if abs(x(2)) == 0,
      disp('Initialized stuck ...');
      phi = 0; % => mode = 0
   else
      disp('Initialized moving ...');
      phi = sign(x(2)); % => mode = +/-1
   end
   xdot = []; reset = []; % to prevent matlab 5 warning msgs!!
   return
end
% model parameters
   m_1 = 5.0;    % Nm/v
   delta = 0.5;  % v
   m_2 = 1.0;    % Nm/v
   f_v = 0.1;    % Nms/rad
   f_c = 1.0;    % Nm
   MoI = 0.01;   % kg-m^2
% first, some preliminary calculations:
   % saturate the input voltage -> electrical torque:
   volts = v_in*sin(omega*t);
   av = abs(volts); sv = sign(volts);
   if av < delta, T_e = m_1*volts;
   else T_e = (m_1*delta + m_2*(av-delta))*sv; end
% now, either execute the model or reset function:
icall = max(abs(imag(mode)));
if icall == 0, % models for dynamic equations and switching funs
   if abs(mode) == 1,  % motor moving:
       xdot(1) = x(2);
       xdot(2) = (T_e - f_v*x(2) - f_c*mode)/MoI;
```

```
        phi = x(2);
    else,               % motor stuck:
        xdot(1) = 0.0;
        xdot(2) = 0.0;
        % NB! phi == 0 until torque suffices for break-away
        if (abs(T_e) - f_c) < 0, phi = 0;
        else phi = (abs(T_e) - f_c)*sign(T_e); end
    end
    reset = []; % to prevent new warning msgs!!
elseif icall == 1, % corresp to state and/or phi reset
    if abs(real(mode)) == 1 % we are going into "stuck" mode...
        reset(1) = x(1);
        reset(2) = 0.0;  % do this so we EXACTLY stick
        phi = 0;  % => mode = 0
    else % we are leaving "stuck" mode
        reset = [];  % no need to reset states...
        phi = sign(T_e);
    end;
    xdot = []; % to prevent new warning msgs!!
else
    error('bad value of mode')
end
```

The initialization part of this model is similar to that in the relay model above. The added complexity in using the more advanced features of this example arise first in the need to distinguish between the model being called for *evaluating* $\dot{x}$ *and* S (`xdot` and `phi`) and being called for *state reset* (`reset`). The state-event handling routines signal this by making `mode` *complex-valued* when state reset is being requested, specifically `mode_reset = mode + jay` where `jay` $= \sqrt{-1}$. Within the code for evaluating $\dot{x}$ and S we have two cases ("stuck" and "moving"), which is quite transparent. The section for state reset similarly has two cases, one in which state reset is actually implemented (when going into "stuck" mode) and one where there is no need to reset (when leaving "stuck" mode); again, the code is quite simple. We note, however, that the reset code for `phi` needs the variable `T_e`, which requires that the preliminary calculations be done outside the model section, otherwise MATLAB will grumble that `phi` has the wrong dimension (because it is returned with the value "empty").

3 Extended Integration Schema

A second significant extension must be made in the MATLAB numerical integration algorithms: neither those in MATLAB, i.e., `ode23` and `ode45`, nor those in SIMULINK, i.e., `gear`, `rk23` and `rk45`, can handle state events in the desired fashion. There are three features needed to permit the MATLAB integration routines to deal with state events:

1. the numerical integrator must coordinate with the extended model to set the initial value of `mode`,
2. it must continuously test for the occurrence of state event(s) by watching for zero crossings in the switching variable(s), and
3. the routine should permit state variables to be reset at a state event, in a rigorously prescribed manner.

MATLAB code for such integrators may be inspected by listing files `trap_101.m` (a simple fixed-step "trapezoidal" integration method) and `ode45_101.m` (an extension of MATLAB's `ode45` variable-step Runge-Kutta-Fehlberg integrator). Both files are provided with a reasonable level of commenting, for your guidance.

4 Example Applications

Two examples used in testing the above integration approach have been described previously; these are an elementary switching relay system [7] and a double-switching system with state reset [8]. Both of these are demonstrated in the files included in this distribution. In addition, we include a switching relay with deadzone, to illustrate the use of "on-boundary" event modeling (a case where mode = 0 for a period of time); an electro-mechanical system with stiction and saturation, to provide further insight into using the more advanced features of this framework; and finally a still more realistic missile roll-control problem from Gibson [10].

4.1 Relay Switching System

The following state equations define the simple relay switching system:

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= -\text{sign}(x_1) \end{aligned}$$

The extended MATLAB model (Fig. 1) for this system is provided above (Section 2.1). The macro `run_relay` demonstrates handling this model using `trap_101` and compares those results with a run using SimuLink [9]; we note that the `rk45` algorithm from SimuLink seems to be particularly inaccurate with this model, due to missing the state events by a large margin[2].

4.2 Relay with Deadzone Switching System

The model for the relay switching system is modified slightly by the introduction of a deadzone,

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= -\text{rdz}(x_1) \end{aligned}$$

[2] In MATLAB 5 this comparison can't easily be made, since running SimuLink from a command line is no longer permitted.

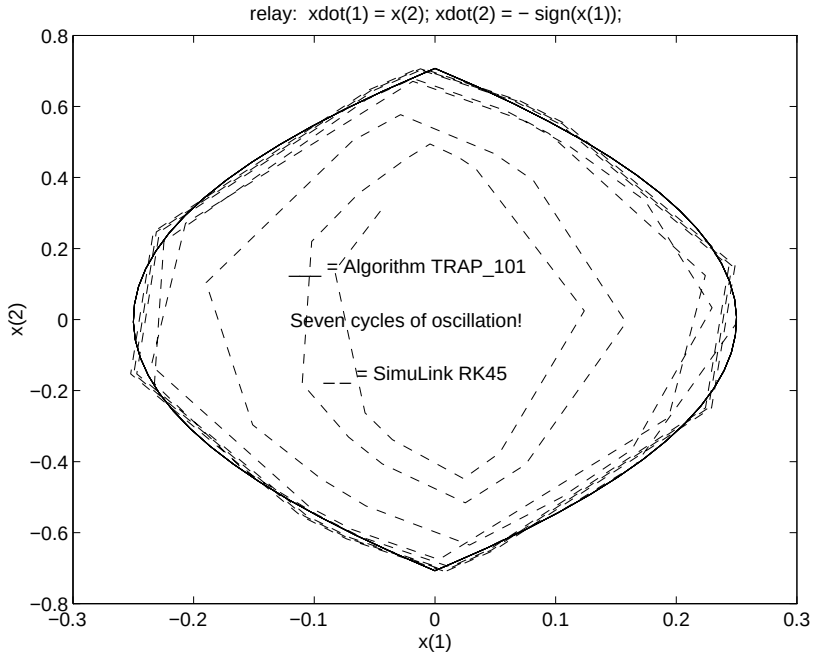


Figure 2: State-event Handling, Simple Relay System

where

$$\mathrm{rdz}(x_1) = \begin{cases} \mathrm{sign}(x_1)\ , & |x_1| > 1 \\ 0 & \text{otherwise} \end{cases}$$

In modeling this component, we have used one mode and the concept of "on-boundary events", so `mode` $= -1$ corresponds to $x_1 < -1$, `mode` $= +1$ corresponds to $x_1 > 1$, and `mode` $= 0$ corresponds to $|x_1| \leq 1$; this means that the definition of the switching function depends on `mode`. We could have used two switching boundaries and modes, but the single-mode approach is presented as an illustration and for its elegance.

The macro `run_rdz` again shows the integration of this model using `trap_101` and compares the results with a run using MATLAB; we note that the `ode45` algorithm seems to be quite accurate with this model, but is very susceptible to "creeping", i.e., reducing the stepsize to a very small value to handle integration errors associated with discontinuities (state events) that are not signaled and captured as in our scheme. To conserve space, this result is not shown; please execute `run_rdz` to see the plots and timing comparisons.

4.3 Twin Relays with State Reset

For this example we have two uncoupled systems of the form:

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= -\mathrm{sign}(x_1) \end{aligned}$$

and similarly for x_3, x_4, with state events defined by

$$S_1 = x_1\ ; \quad S_2 = x_3$$

and a reset definition akin to the bounce of a ball with coefficient of restitution 0.8,

$$x_c(t_e^+) = \mathrm{col}\,[\, x_1(t_e^-) \quad 0.8 \times x_2(t_e^-) \quad x_3(t_e^-) \quad 0.8 \times x_4(t_e^-)\,]$$

(assuming both switchings occur simultaneously). This model presents several challenges: It is easy to pick initial conditions that make the state events simultaneous, and there is always a finite time when the switching rate becomes infinite. In the simulations executed by the macro `run_rdz`, we choose the initial condition $x(0) =$ `[ 0.25 ; 0 ; -0.25 ; 0 ]` so the state events are simultaneous and the switching times t_k have the limiting value $t_\infty = 6.36396$[3]. Note that we cannot compare integration accuracy or times with standard MATLAB integrators in this example as we did above, because they cannot handle a problem with state reset.

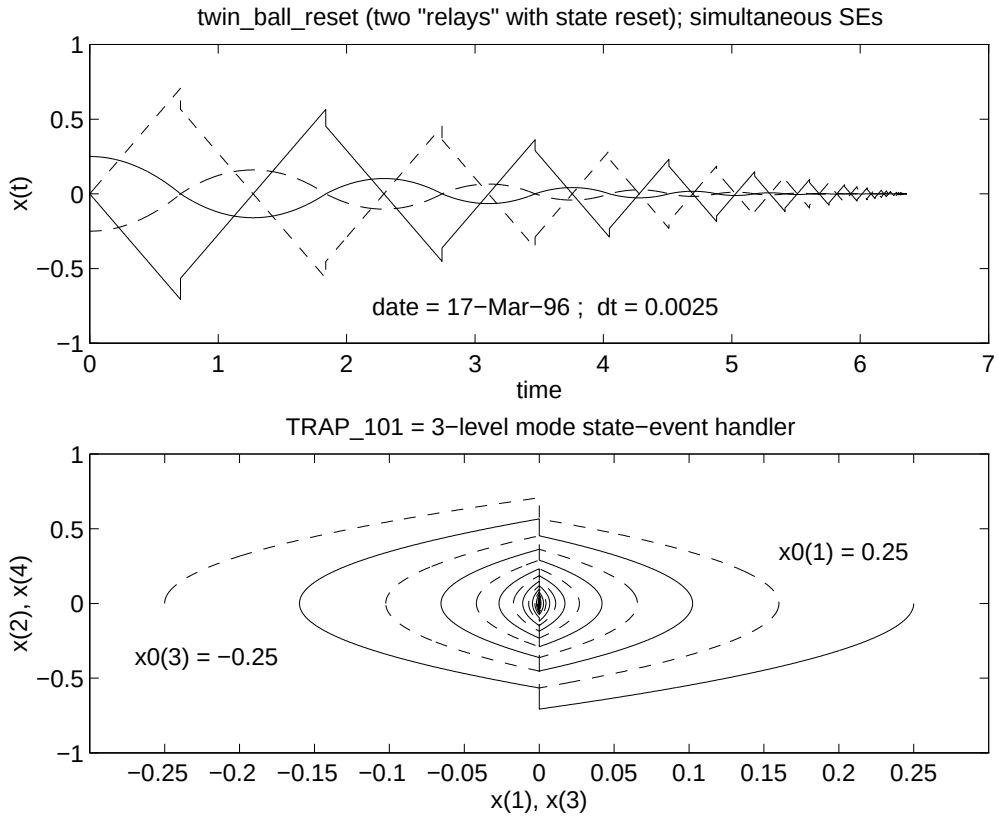


Figure 3: State-event Handling, Twin Relays with Reset

[3]If the reset coefficient is $\alpha = 0.8$ then the switching time t_k for one of the uncoupled systems for initial condition `[ x_0 ; 0 ]` is $t_k = 2\sqrt{2\mathtt{x\_0}}\{1/2 + \alpha + \alpha^2 + \ldots + \alpha^{k-1}\}$, so

$$t_\infty = \sqrt{2\mathtt{x\_0}}\,\frac{1+\alpha}{1-\alpha}$$

4.4 Electro-mechanical System Model

The following state equations define the dynamics of an electro-mechanical system with saturation and stiction:

$$
\begin{aligned}
\dot{x_1} &= x_2 \\
\dot{x_2} &= T_m/J \\
T_m &= \text{stick}(T_e, x_2) = \begin{cases} T_e - f_v x_2 - f_c \text{sign}(x_2) & , x_2 \neq 0 \text{ or } |T_e| \geq f_c \\ 0 & , x_2 = 0 \ \& \ |T_e| < f_c \end{cases} \\
T_e &= \text{sat}(v) = \begin{cases} m_1 v & , |v| < \delta \\ (m_1 \delta + m_2(|v| - \delta))\text{sign}(v) & , |v| \geq \delta \end{cases}
\end{aligned}
$$

where T_e is the "electrical torque", a saturated function of the input voltage v, T_m is the stiction model, and the physical parameters are as defined in the model code, e.g., $J =$ `MoI` (section 2.2). The macro `run_stick` demonstrates running this model using `trap_101` and compares those results with a run using MATLAB's `ode45` integrator; we note again that this algorithm gives rise to the "creeping solution" problem unless the tolerance is substantially reduced (from 1.e-06 to 0.005 in this macro); unfortunately this results in rather undesirable "chattering" behavior that is readily apparent in the simulation plots where the motor should be stuck (e.g., $t \approx 1.6$ and 4.8 seconds).

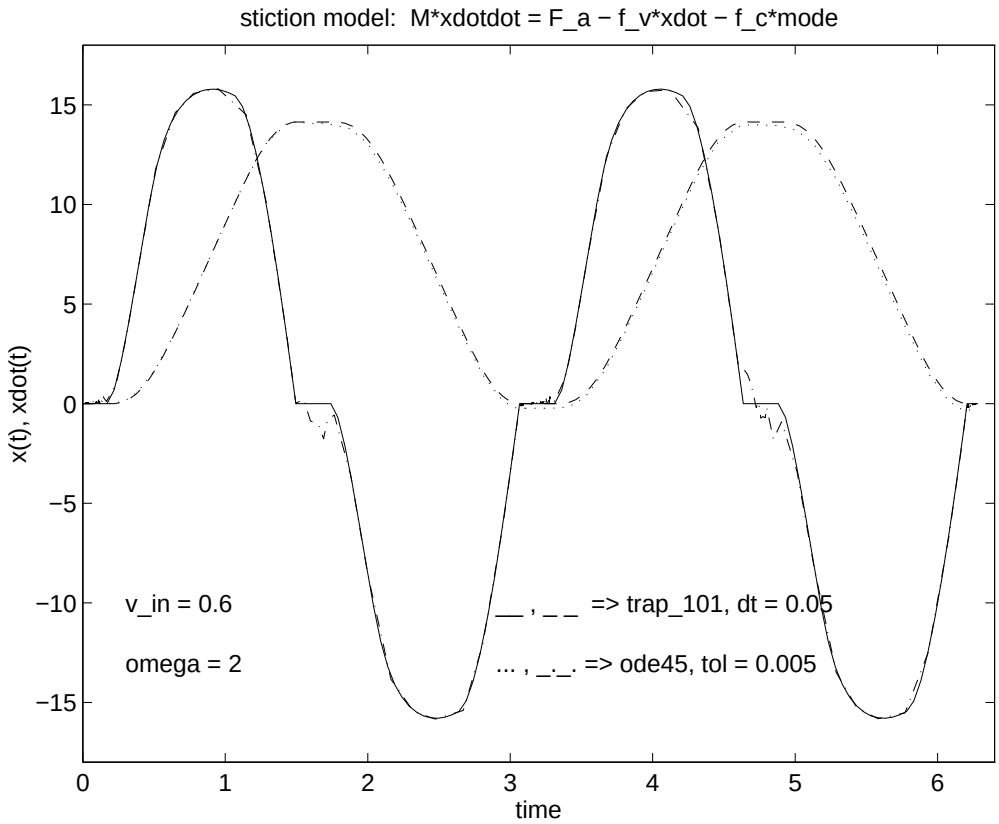


Figure 4: State-event Handling, Electro-mechanical System

4.5 Missile Roll-Control System Model (Gibson)

The following problem is posed by Gibson [10], and a describing-function solution for limit cycle conditions is provided in [11]. We repeat the problem formulation from the latter citation (converted to SI units):

Assume a pair of reaction jets is mounted on the missile, one to produce torque about the roll axis in the clockwise sense and one in the counterclockwise sense. The force exerted by each jet is $F_0 = 445$ N and the moment arms are $R_0 = 0.61$ m. The moment of inertia about the roll axis is $J = 4.68$ N·m/sec^2. Let the control jets and associated servo actuator have a hysteresis $h = 22.24$ N and two lags corresponding to time constants of 0.01 sec and 0.05 sec. To control the roll motion, there is roll and roll-rate feedback, with gains of $K_p = 1868$ N/radian and $K_v = 186.8$ N/(radian/sec) respectively. The block diagram for this system is shown in Fig. 5, and the model is listed in the Appendix.

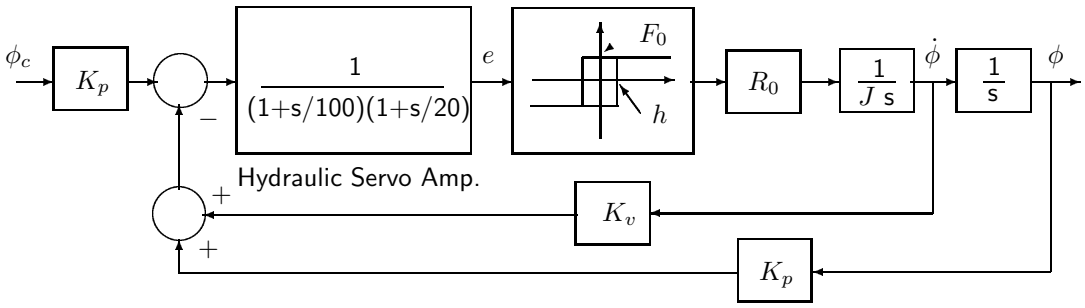


Figure 5: Block Diagram, Missile Roll-Control Problem [Gibson]

A highly rigorous digital simulation using `ode45_101` to capture the switching characteristics of the hysteretic relay yielded $A_{LC} = 0.130$, $\omega_{LC} = 23.1$ rad/sec, as shown in Fig. 6. In Gibson, it is said that an analog computer solution yielded $A_{LC} = 0.135$ rad and $\omega_{LC} = 22.9$ rad/sec, which agrees quite well with the result depicted in Fig. 6; the describing function solution [11] confirms these simulations, giving $A_{LC} = 0.124$ rad, $\omega_{LC} = 24.36$ rad/sec.

5 Conclusion

The methodology, algorithms and examples presented above provide a direct demonstration of our approach and software for state-event handling in continuous-time components. We have applied this technique to models of several electro-mechanical systems, including those exhibiting backlash and stiction, and in every case we have obtained excellent results (accurate solutions without excessive computational burden). These algorithms are being extended to deal with more comprehensive hybrid systems, i.e. those with both continuous- and discrete-time components [12].

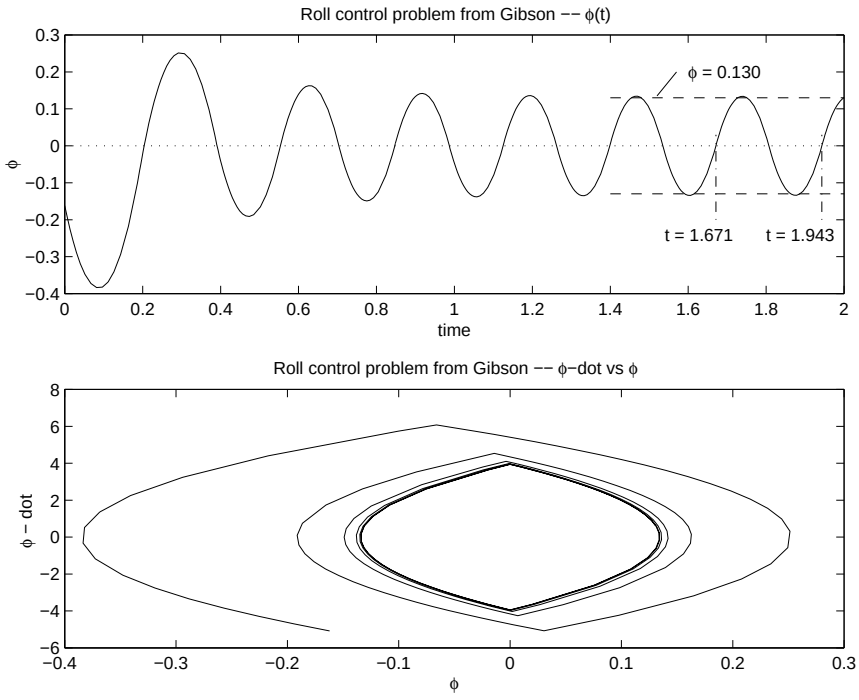


Figure 6: Missile Roll-Control Simulation Result

This brief tutorial is intended to supplement the demonstration models and macros provided with this distribution. These models may be studied and the macros executed to learn more about the features of this software. If you have problems or questions about the use of these integration routines, please let us know and we will *try* to supply further assistance.

References

[1] Taylor, J. H. "Toward a Modeling Language Standard for Hybrid Dynamical Systems", *Proc. 32nd* IEEE *Conference on Decision and Control,* San Antonio, TX, December 1993.

[2] Taylor, J. H. "A Modeling Language for Hybrid Systems", *Proc.* IEEE/IFAC *Symposium on Computer-Aided Control System Design*, Tucson, AZ, March 1994.

[3] Elmqvist, H., Cellier, F. E. and Otter, M., "Object-Oriented Modeling of Power-Electronic Circuits Using Dymola", *Proc. CISS'94* (First Joint Conference of International Simulation Societies), Zurich, Switzerland, August 1994.

[4] Cellier, F. E., Elmqvist, H., Otter, M. and Taylor, J. H., "Guidelines for Modeling and Simulation of Hybrid Systems", *Proc.* IFAC *World Congress*, Sydney Australia, 18–23 July 1993.

[5] Cellier, F. E., Otter, M. and Elmqvist, H., "Bond Graph Modeling of Variable Structure Systems", *Proc. ICBGM'95* (Second International Conference on Bond Graph Modeling and Simulation), Las Vegas, Nevada, January 1995.

[6] MATLAB *User's Guide*, The MathWorks, Inc., Natick, MA 01760.

[7] Taylor, J. H. "Rigorous Handling of State Events in MATLAB", *Proc. 4th (*IEEE *Conference on Control Applications,* Albany, NY, 28–29 September 1995.

[8] Taylor, J. H. and Kebede, D., "Modeling and Simulation of Hybrid Systems", *Proc. IEEE Conference on Decision and Control*, New Orleans, LA, 13–15 December 1995.

[9] SIMULINK *User's Guide*, The MathWorks, Inc., Natick, MA 01760.

[10] J. E. Gibson, *Nonlinear Automatic Control*, McGraw-Hill Book Co., New York, NY, 1963.

[11] J. H. Taylor, "Describing Functions", article No. 2409 in *Electrical Engineering Encyclopedia,* John Wiley & Sons, Inc., New York, 1999.

[12] Taylor, J. H. and Kebede, D., "Modeling and Simulation of Hybrid Systems in MATLAB", *Proc. IFAC World Congress*, San Francisco, CA, July 1996.

[13] Taylor, J. H. *A Rigorous Modeling and Simulation Package for Hybrid Systems*, US National Science Foundation SBIR Report, Award No. III-9361232, Odyssey Research Associates, Inc., June 1994 (available only from the author).

Appendix: Model for Gibson's Missile Roll-Control Problem

```
function [xdot,phi,reset] = relay(t,x,mode)
%% version of Gibson roll-control problem suitable for
%% simulation via ode45_101 (using modes)
%
% JH Taylor - 24 March 1999

%%% converted to SI units
% states x1 = phi, x2 = phidot, x3, x4 = servoamp model states
% define the servoamp dynamics (convert transfer fn to state space)
% A = [ 0 1 ; -2000 -120 ]; B = [ 0 ; 1 ]; C = [ 2000 0 ]; D = 0;

h = 22.24; F0 = 444.8; % thruster "relay" constants (Newtons)
J = 4.68; % N-m/sec^2; = 3.45 lb-ft/sec^2
R0 = 0.6096; % m; = 2 ft
Kp = 1868; % N/rad; = 420 lb/rad
Kv = 186.8; % N/(rad/sec); = 42 lb/(rad/sec)
```

```
% initializes the mode in ode45_101
if isempty(mode);
  e = 2000*x(3);
  if e > h, phi(1) = 1.0;
  elseif e < -h, phi(1) = -1.0;
  else error('sorry - cannot initialize properly . . .');
  end;
  xdot = [];
  reset = [];
  return;
end;

% actual model and switching logic
e = 2000*x(3);
icall = max(abs(imag(mode)));
if icall == 0, % dynamic equations and switching funs
   xdot(1) = x(2);
   thrust = F0 * mode;
   xdot(2) = R0*thrust/J;
   xdot(3) = x(4);
   Uamp = - Kv*x(2) - Kp*x(1);
   xdot(4) = -2000*x(3) -120*x(4) + Uamp;
   if mode == 1, phi(1) = e + h;
   elseif mode == -1, phi(1) = e - h;
   else error('sorry - mode should never be 0!!');
   end
   reset = [];
elseif icall == 1, % corresp to possible state reset
   xdot = [];
   phi(1) = NaN;
   reset = [];
   disp(['switching at t = ',num2str(t)])
else
   error('bad value of mode')
end
% end of relay with hysteresis model
```