# A Creation of Hybrid System Modeling and Simulation Environment in matlab

by

Jie Zhang
BScE(EE), Nankai University, China, 2002

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of

Master of Science and Engineering

in the Department of Electrical and Computer Engineering

| | |
|---|---|
| Supervisor: | James H. Taylor, Ph.D., Electrical and Computer Engineering |
| Examining Board: | R. Doraiswami, Ph.D., Electrical and Computer Engineering<br>C. Diduch, Ph.D., Electrical and Computer Engineering<br>R. Dubay, Ph.D., Mechanical Engineering |

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2005

# Abstract

Modeling and simulation of hybrid systems is an important procedure in control system design and applications. A few commercial software tools offer a decent solution for this need. However, the algorithms are not efficient, even inaccurate, for some complicated dynamical systems. In this thesis, a new approach is introduced for the modeling and simulation of hybrid systems. A modeling and simulation environment is built within MATLAB. It utilizes a medium order variable step size numerical integration algorithm and an advanced state event handler to solve ordinary differential equations. It also includes a rigorous time event handler for discrete-time parts of the system. It is a general, accurate and efficient tool for modeling and simulation of hybrid systems. Some examples to show these advantages are given in this thesis.

# Acknowledgments

I would like to take this space to thank Dr. James H. Taylor for all his help and his patience throughout my research process.

I also would like to thank Hong Tang for her advice on my thesis.

I also thank all friends who gave me help and advice.

# Contents

# List of Figures

# Nomenclature

Abbreviations

| | |
|---|---|
| CTC | continuous-time component |
| DTC | discrete-time component |
| LBC | logic-based component |
| SHSML | Standard Hybrid Systems Modeling Language |
| HSML | Hybrid Systems Modeling Language |
| ODE | ordinary differential equation |
| DE | difference equation |

Symbols

| | |
|---|---|
| $x, x_c$ | CTC state variable |
| $\dot{x}, \dot{x_c}$ | derivative of the CTC state variable |
| $t$ | time |
| $u, u_c$ | CTC input |
| $y, y_c$ | CTC output |
| $x_{d,k+1}, x_{d,k}, x_d$ | DTC state variable |
| $y_{d,k}, y_d$ | DTC output |
| $u_k, u_{d,k}$ | DTC input |

| | |
|---|---|
| $t_k$ | DTC time sequence |
| $k$ | DTC sampling index |
| $x_0, t_0$ | initial conditions |
| $h$ | integration step size |
| $m, mode$ | CTC integration mode variable |
| $t_e$ | DTC next execution time |
| $ndtc$ | DTC execution flag variable |
| $\phi, phi$ | switching function |
| $r, reset$ | CTC state variable reset value |
| $sysout$ | user-defined output variable |

# Chapter 1

# Introduction

## 1.1 Dynamical Control System Simulation

Modeling and simulation of dynamical control systems is a well established technical discipline. The corresponding activity is widespread throughout the scientific and engineering fields. Most dynamical control systems include nonlinear and computer-controlled subsystems. For example, computer-controlled dynamical systems include modern automobiles, appliances, automatic teller machines and so on; in higher technology areas they encompass flexible manufacturing systems, robots, intelligent vehicles and highway systems, spacecraft, flight dynamics and control, chemical and materials process control, automated drug administration and health monitoring systems, to name only a few more recent applications.

Modeling and simulation of a dynamical system may be used to predict the behavior

of a device that is in the early design stages of the development. It may also be used to refine the design of a system, which has been prototyped or is targeted for redesign, or to perform experiments that are difficult or impossible to execute in the real world, such as studying failure modes in a flight control system or nuclear reactor control logic.

Computer-controlled dynamical systems have significantly advanced in recent decades. They have been developed and implemented in many areas. The physical components, such as airframe, robot and chemical process, may not be much more complicated, but the discrete-time components and computer control software have been developed to a more complex and higher level. These developments have resulted in a large surge of interest in "hybrid systems", as the modern class of computer-controlled dynamical systems has come to be called.

## 1.2 Hybrid Systems

The term "hybrid systems" is not yet sufficiently well understood in the literature to mean one and only one specific article. A precise definition of the term "hybrid system" must be provided before we sets out to define and implement a modeling and simulation environment for such systems.

Based on the literature related to hybrid control systems, we can give a definition for the term "hybrid systems" as follows: Hybrid systems are systems described,

either during the whole period under investigation or during a part of it, by a fixed or variable set of differential equations where at least one state variable or one state derivative is not continuous over a simulation run [2].

In a more straightforward way, the term "hybrid systems" can also be expressed as: Hybrid systems are composed by interconnecting continuous-time components (CTCs), discrete-time components (DTCs), and logic-based components (LBCs) in some arbitrary configuration [16].

CTCs can be expressed using Ordinary Differential Equations (ODEs). DTCs can be expressed using Difference Equations (DEs). LBCs have no "generic form" in mathematical terms. In fact, the term "DTC" mentioned previously can be considered to be a special case of an LBC in some sense (e.g., both are realized in software). However, we use DTCs for well-defined and commonplace discrete-time numerical algorithms as DEs and LBCs for the components that are primarily logical or symbolic in nature.

## 1.3 Motivation

Modeling and simulation of hybrid systems is a challenging and complex task, which has been gaining attention. Hybrid systems have became more and more common and important in the real life. "Controlling with a digital computer is of growing importance in many fields. The use of computers as a control device is attractive mainly because of offering flexibility of the control programs and the decision-making

capability of digital systems which can be shared with control functions to meet other system requirements [14]." Furthermore, modern hybrid control systems always consist of a continuous plant under the control of a discrete-time system [12].

The problem is that "much of the work in modeling and simulation is being done today with outmoded and possibly dangerous tools" [16]. Most commercial software packages describe continuous-time systems using ODEs that are the functions of state variable and time (e.g., the 'S-function' in SIMULINK). To introduce a notation, a continuous-time system is expressed as:

$$\dot{x} = f(t, x, u)$$

$$y = g(t, x, u)$$

where $x$ is the state variable, $t$ is the time, $u$ is the input to the system, and $y$ is the system output. This modeling and simulation technology may be dangerous if the derivative function $f$ is different when the system is running in different modes. The simulation software is not able to identify different stages correctly only from $t$, $x$ and $u$. Mode identification failure will cause an incorrect simulation result.

Moreover, the simulation software without the mechanism for handling the mode change mentioned above could be inefficient for some dynamical systems (e.g., a DC motor that has "stiction" or friction with sticking when velocity passes through zero).

All the trends illustrated above establish a requirement for a rigorous and efficient

modeling and simulation environment for hybrid systems. A lot of research in this field has been done in the past few decades, but more remains to be done.

In 1979, Mr. Francois Edouard Cellier developed a simulation method for hybrid systems [2]. The method provided a algorithm for the state event handling. However, this method was applied in Fortran in the early stage, so the user needs strong programming skills. Thus, few people use this method today when commercial software has become popular.

In 1993, Dr. Taylor creates a standard hybrid systems modeling language (SHSML), which provided a rigorous means for modeling and evaluating hybrid systems [15].

Tsybatov, V. and Vittikh, V. created "General Modeling System (GMS)" and "Natural Simulation Language (NSL)" [23], which are still in the conceptual stage. Also, Petri Nets (PN) is a powerful tool presented in the paper written by Rezai, M. in 1995 for the modeling of systems exhibiting concurrency and synchronization characteristics [10]. Symbolic and interval methods for simulation of hybrid dynamic systems are presented by Nedialko S. and Mohrenschildt [8]. Other related research can be found in [7, 6, 5, 11, 22, 13]. The research mentioned above does not concentrate on rigorous state event handling. So, those approaches did not solve the problem.

Recently, Dr. Taylor began to develop a rigorous modeling and simulation environment within MATLAB [18]. It can be considered as an implementation of "Hybrid System Modeling Language (HSML)" [16] which is also created by Dr. Taylor as an

extension of SHSML. Since programming in MATLAB is very common for most engineers, such environments become more and more user-friendly, and it is efficient due to the use of advanced algorithms. So far, the environment development is still an on going project [17]. The main disadvantage of the environment in [17] is that it cannot handle discrete-time components. Thus, it cannot be used for hybrid systems. This thesis is a continuation to extend the environment to be capable of handling hybrid systems based on Dr. Taylor's previous research, which included a "road map" for this research [18, 20, 21].

The goal of this project is the creation of a rigorous, portable and "user-friendly" environment for modeling and simulating hybrid systems. This will encompass both physical processes with discontinuities and discrete-time control algorithms. Such an environment is the key for research and development activities in hybrid control: If hybrid systems cannot be simulated correctly, then all attempts to verify system designs and behavior are open to question.

# Chapter 2

# Background Review

Before introducing the environment developed by this thesis, we will briefly review several related formulations and algorithms.

## 2.1 Continuous-time System Model Formulation

Most physical systems are modeled as continuous-time systems, since they can be described using Ordinary Differential Equations (ODEs).

$$\dot{x}_c = f_c(x_c, u_c, t)$$

$$y_c = g_c(x_c, u_c, t)$$

where $x_c$ is the state vector, $y_c$ is the output vector, $u_c$ is the continuous-time input signal, and $t$ is the time.

## 2.2 Discrete-time System Model Formulation

Discrete-time systems can be normally represented using Difference Equations (DEs).

$$x_{d,k} = f_k(x_{d,k-1}, u_{k-1})$$

$$y_{d,k} = g_k(x_{d,k}, u_k)$$

where $x_{d,k}$ is the discrete state vector, $k$ is the index corresponding to the discrete time point $t_k$, $y_{d,k}$ is the output vector, and $u_k$ is the input of discrete-time system.

## 2.3 Algorithms to Solve ODEs

Simulating continuous-time systems involves solving their ODEs for a given initial condition $x_0(t_0)$ and input $u_c(t), t \geq t_0$. From a mathematical point of view, solving ODEs can be done analytically by performing Laplace Transform on linear ODEs. However, for general nonlinear systems analytic solution is not possible, so a computer solves ODEs in a different way. It involves a series of numerical integration steps. Each point in the trajectory is calculated from the derivatives that are provides by the ODEs. The algorithms for solving ODEs include the Euler algorithm, the modified Euler

algorithm, the Runge-Kutta algorithm and so on. The Runge-Kutta algorithm is a medium order method. It is known to be a very accurate and well-behaved algorithm for a wide range of problems.

The Runge-Kutta family of algorithms provides a method to solve a differential equation numerically (i.e., approximately). Considering a single variable problem

$$\dot{x} = f(t, x)$$

with initial condition $x(0) = x_0$. Assume $x_n$ is the value of the variable at time $t_n$. A Runge-Kutta formula takes $x_n$ and $t_n$ and calculates an approximation for $x_{n+1}$ at a brief time later, $t_{n+1} = t_n + h$. It uses a weighted average of values of $f(t, x)$ at several times within the interval $(t_n, t_n + h)$. One standard formula [9] is given by

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1)$$

$$k_3 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2)$$

$$k_4 = f(t_n + h, x_n + hk_3)$$

To run the simulation, the computer simply starts with $x_0$ and find $x_1$ using the above formula, then it uses $x_1$ to find $x_2$ and so on.

The integration step size $h$ should be small enough to ensure the calculation accuracy. The smaller the step is, the more accurate the result is and the more time it will take. If $h$ is given before the integration process and fixed during the process, the algorithm is called a fixed step size algorithm. If $h$ can be varied by the algorithm during the integration process, it is called a variable step size algorithm. The fixed step size algorithm is easy to perform, but it has some disadvantages, which include that it is not accurate enough when the system has a fast behavior and more time consuming when the system has a slow behavior. Therefore, for different ODEs (systems), the step should be different depending on how fast the system changes. Even in one system, the step size may need to change from time to time to optimize the balance between the accuracy and the execution time.

Thus, most simulation software uses variable step size algorithms instead of fixed step size algorithms. The common examples of the variable step size algorithms include "ode45" and "ode23" routines in MATLAB. The solver "ode45" applies variable step size integration algorithm based on the Runge-Kutta-Fehlberg formula. It adjusts step size by calculating the error. It is a very efficient and accurate solver for most of non-stiff systems and systems with continuous dynamics.

## 2.4 Difficulty of Modern Simulation

Although the solvers like "ode45" are perfect for most continuous systems, they are not efficient for nonlinear systems with discontinuities and with discrete-time components. Nonlinear systems and discrete systems have sudden changes in the system behavior. The routine 'ode45' is invoked in the following form:

```
[T,X] = ODE45('ODEFUN',TSPAN,X0)
```

With TSPAN = [T0 TFINAL], it integrates the system of differential equations

$$\dot{x} = f(t, x)$$

from time T0 to TFINAL with initial conditions X0. The ODEs are written as a MATLAB function in a file named 'ODEFUN.m'. It still does not answer the problem mentioned in Section 1.3, if the derivative function $f$ does vary depending on the system modes.

A major difficulty in simulation is handling the sudden changes (discontinuities). The sudden changes are called events, and two different cases can be distinguished according to the nature of their occurrence [4].

*State Events*: The events which are produced when the continuous subsystem state reaches some switching condition are called *State Events*. For example, a system with a relay is a nonlinear system. The sudden change happens when the output of the

relay switches. When the input of an ideal relay varies crossing zero, the output of relay will change the sign immediately. The switching of a relay is a state event.

*Time Events*: The events which occur at a given time, independently of what happens in the continuous state, are called *Time Events*. An example could be a system with a digital controller, which is a DTC. The output of a DTC is not continuous either; the sudden change exists at every sampling point.

The previous environment of Dr. Taylor and Kebede [17] can handle state events but not time events. In this thesis, a new mechanism is added for the time event handling so that it is effective for the hybrid systems, as suggested and planned in [18, 20, 21].

# Chapter 3

# Software Framework Design

## 3.1 High-level Design

At the highest level, simulation software is designed as three parts. The basic framework is shown in Figure 3.1.

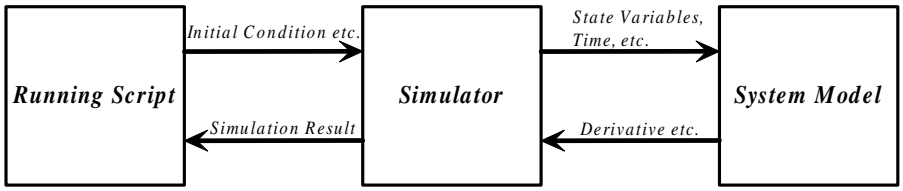


Figure 3.1: The basic framework of the software

The Simulator is the core of the entire simulation environment. It should be able to handle both ODEs and DEs. The algorithm used to solve ODEs is based on the MATLAB routine 'ode45.m', which is a variable step medium order integration solver that works

very well for continuous CTCs. In previous work [17], a state event handler had been added to make the Simulator more efficient for state event handling. In this thesis project, a time event handler has been added so that the Simulator can handle time events (i.e., solve DEs). With both the state event handler and the time event handler, the Simulator is efficient for hybrid systems. The development of the Simulator is the main task of entire software design.

The running script is normally a short program in MATLAB. It is written by the user. The necessary contents of the running script include specifying the simulation time, the initial conditions, the system model name and other required parameters. Users can give commands in the running script to perform any other operations such as plotting variables after simulation, depending on their requirements.

The system model is a MATLAB function that represents the dynamical system. It is also created by the user and must be composed under certain rules so that the Simulator can recognize the system. It plays the same role as the block diagram in SIMULINK. To be convenient, several model templates are given in the software package. Users can build the system model based on an appropriate model template. The main contents that users need to specify are the initialization part, the ODEs for CTCs, the DEs for DTCs if they exist, the state reset part and so on.

Before each system simulation process starts, the running script passes the system model name, the simulation time, the initial conditions, etc. to the Simulator. Then, the Simulator begins the simulation process. During the simulation process, data is

continually transferred between the Simulator and the system model. These data flows are shown in Figure 3.2. After finishing the simulation process, the Simulator returns the result to the running script. Finally, post-simulation operations (e.g., plotting figures) that are defined by users in the running script will be performed.

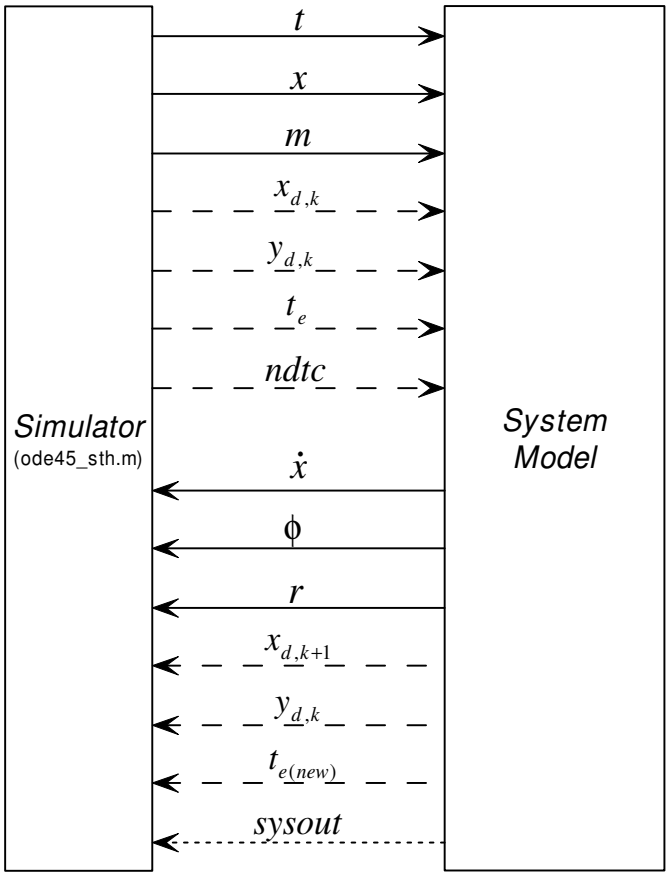


Figure 3.2: The interface between the solver and the system model

Figure 3.2 indicates the interface between the Simulator and the system model, where $t$ is the time, $x$ is the state vector for CTCs, $m$ is the mode vector, $x_d$ is the state vector for DTCs, $y_d$ is the output vector for DTCs, $t_e$ is the DTC execution time, $ndtc$

is the flag variable for invoking DTCs, $\phi$ is the switching function for state event handling and $r$ is the CTC state reset vector. The parameters indicated by solid lines are mandatory for all system models. The parameters indicated by long-dashed lines are optional parameters that are only necessary for systems that contain DTCs. The variable *sysout* that is indicated by a short-dashed line is called the user-defined output variable; it is used to record any system internal values the user may want to access; it is optional whether or not DTCs are included. It can be a vector and is optional for any system.

The interface structure is designed based on the "New MATLAB model component input/output structures", which was created by Dr. Taylor and Mr. Kebede [18, 20, 21]. The new components in the interface are the flag variable *ndtc*, the DTC output vector $y_d$ and the user-defined output variable *sysout*.

## 3.2 System Model Framework Design

In order for the Simulator to be able to recognize and run the system model, the system model must be composed in a rigorously defined form. Therefore, it is necessary to design the system model framework before the Simulator design.

In previous work, the model was designed as two main parts:

- Initialization (e.g., setting the correct modes)
- Calculations for the simulation process (e.g., derivatives, switching functions,

reset values)

Variable *mode* is used to indicate which stage the CTC model is in for the current integration step. It is a critical variable used to handle the state events. The state event handling will be introduced specifically in Chapter 4. The initialization part returns the initial switching function $phi(\phi)$. The variable *mode* is initialized as $mode = sign(phi)$.

In order to show the relation between *mode* and *phi*, an example is given as follows: Figure 3.3 shows the behavior of a relay with deadzone, where $y$ is the relay output

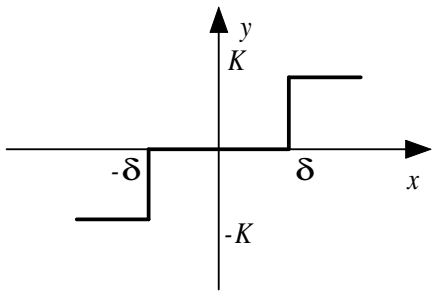


Figure 3.3: A relay with deadzone

and $x$ the is the relay input. In this example, variable *mode* can be used to represent the on/off stage of the relay. When $y = -K$, the relay is reversed, $mode = -1$, and the switching function of the relay is $phi = x + \delta$, i.e., the relay will switch when $x$ passes through the value $-\delta$. Similarly, when $y = K$, the relay is on, $mode = 1$, the switching function of the relay is $phi = x - \delta$. Finally, when $-\delta < x < \delta$ the relay is off and $mode = 0$. The initialization section sets *phi* based on the initial condition $x_0$ so the *mode* is consistent.

In the calculation part, the system model calculates the derivatives and the switching function. This part has two sections. One is used in regular integration process. The other one is used to determine what to do when state events happen.

Since the new software has DTC handler (time event handler), there should be another calculation part in the system model exclusively for DTC updating. Therefore, the new model framework was designed as:

1. Initialization
2. Calculation for CTCs for the simulation process
   a. Derivative and switching function evaluation
   b. State reset at state events
3. Calculation for DTCs for the simulation process

The initialization and calculation for CTC parts have the same structure as the previous model framework. The calculation for DTC part is used for DTCs update.

A system model template that corresponds to Figure 3.2 is included in the software package, as shown below:

```
function
[xdot,phi,reset,xdp,yd,tep,sysout]=system_model(t,x,mode,xd,yd,te,ndtc)


%initialization, executed with mode=[](empty)
```

```
if isempty(mode) == true,
    phi = ?; % determined by x(t0)
    xdot = [];
    reset = [];
    xdp = ?;
    yd = ?;
    tep = ?; % cannot be smaller or equal to t0
    sysout = ?; %Auxiliary system output evaluation
    return;
end

%CTC derivative, switching function and reset calculation
rf = max(abs(imag(mode)));
if rf == 0, % set derivative and switching function when mode is real
    xdot = ?;
    phi = ?;
    reset = [];
    xdp = xd;
    yd = yd;
    tep = te;
else  % set CTC state reset value when mode is complex
    xdot = [];
```

```
    phi = ?;

    reset = ?;

    xdp = xd;

    yd = yd;

    tep = te;

end


% update DTCs ready to be executed

for n = 1:length(ndtc)

    if ndtc(n) == true,

        tep(n) = ?;

        xdp(?) = ?;

        yd(n) = ?;

    end

end


sysout=?; %Auxiliary system output evaluation
```

### 3.2.1 Initialization

In the first section, the initialization part, $\phi$ (*phi*) is the only mandatory parameter that needs to be returned back to the Simulator. The variable $\phi$ is the switching

function corresponding to the initial condition. For example, considering a second order system, which has three different stages, $mode = 1$ refers to stage A, $mode = -1$ refers to stage B and $mode = 0$ refers to motion on the switch line, as shown in Figure 3.4. Since there is only one switching function $S$ no matter which state the simulation

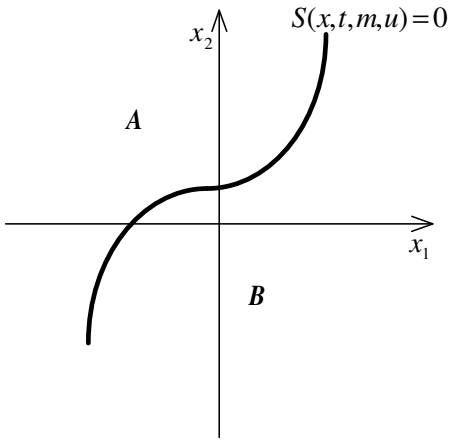


Figure 3.4: A simple example to show the relation between initial *mode* and $\phi$

starts in, *phi* should be assigned to the switching function $S$.

$$\phi = S(t, x, m, u)$$

If the initial point is in stage A, $\phi$ should be a positive number; if the initial point is in stage B, $\phi$ should be a negative number; if the initial point is on the switching line, $\phi = 0$.

If a system has DTCs, the variables for DTCs $x_{d,k+1}$ $(xdp)$, $y_{d,k}$ $(yd)$, $t_{e(new)}$ $(tep)$ need to be assigned also. The values of $x_{d,1}$ and $y_{d,0}$ are derived from the initial conditions

($x_{d,0}$) of the DTCs and the initial inputs ($u_{d,0}$) to the DTCs. The next execution time *tep* for the DTC(s) should also be assigned.

### 3.2.2 CTC model evaluations

In the second part, the CTC state derivatives, switching functions and state resets are calculated. $\dot{x}$, $\phi$ and *reset* need to be assigned. If a system has DTCs, *xdp*, *yd* and *tep* should be written in the exact form as shown in the template.

This part consists two independent sections. The first section is used for derivative and switching function evaluation; the second section is used for reset value evaluation. When *mode* is not complex, i.e., $imag(mode) = 0$, the first section is executed. When *mode* is complex, i.e., $imag(mode) \neq 0$, the second section is executed. The simulator makes *mode* complex to signal the model that it is time to reset the states, if necessary.

Variable *mode* is transferred from the Simulator. It does not become complex until a state event is caught. Therefore, as long as no state event happens, only the first section is executed. In this section, the state derivative vector $\dot{x}$ should be assigned to the proper value; and, the switching function $\phi$ corresponding to the current state and *mode* should be assigned. Variable *reset* can be assigned to any number or empty since it will not be used by the Simulator.

When a state event happens, the Simulator converts *mode* to a complex number to activate the second part to check if there is a reset value. In the second part, if the system has a reset value, *reset* should be assigned to an appropriate value. Variable

$\phi$ can be assigned to any number except 0, unless the state event is an on-boundary event (this will be introduced in more detail in Chapter 4). Variable $\dot{x}$ can be assigned to any number or empty since it will not be used by the Simulator.

### 3.2.3 DTC model evaluations

The third part is used for DTC update. The variable *ndtc* is a vector used to indicate which DTCs need to be updated. The length of *ndtc* equals to the number of DTCs. The elements of *ndtc* are set to 1 if the corresponding DTCs need to be updated.

According to the system model template, if the $N$th DTC should be updated, $ndtc(N) = 1$. Then $tep(N)$ should be assigned to the next execution time for this DTC, and the DTC output $yd(N)$ should also be assigned to the output value. The DTC state vector $xdp$ should also be assigned. However, the index number(s) of $xdp$ is (are) uncertain, which should be specified by the user, as explained below (Section 5.1).

State variables and the derivatives, $x$, $x_{d,k}$, $\dot{x}$ and $x_{d,k+1}$ are column vectors. Their lengths equal to the summation of the order of each CTC or DTC. For example, if a system has two CTCs – one is a second order CTC, and the other one is a third order CTC, the length of $x$ and $\dot{x}$ should be five; if a system has three DTCs – two are first order DTCs, and the other one is a second order DTC, the length of $x_{d,k}$ and $x_{d,k+1}$ should be four. The output of DTCs $y_{d,k}$ and $t_e$ are also column vectors, whose lengthes equal to the number of DTCs. Variable *mode* could be a vector, whose length depends on how many switching functions the system has.

A more detailed description of composing a hybrid system model will be given in Chapter 7.

# Chapter 4

# State Event Handling

The state-event handler is designed to permit more accurate and efficient integration for CTCs that may have discontinuous behavior such as relays switching and mechanical components engaging/disengaging [18]. Without a state-event handler, the difficulty is that there always exists overshoot when the integration step reaches the switching points of discontinuities. Then the integration step size will be reduced and the solver will calculate the error using a specific formula to determine if this revised step size is acceptable. This process will be repeated until the error is smaller than the tolerance (by default $tol = 1e - 6$). So, the switching point for a discontinuity is detected by the error calculation and the accuracy depends on the error tolerance.

Using the error calculation to detect a switching point produces two major problems: first, the switching point may be found after many times of step size reduction. In this situation the simulation process becomes very slow since much simulation time is

consumed around the switching point. Second, the actual switching point can not be located accurately enough. Therefore, for example, the detected switching point in each limit cycle of a relay system may be not exactly determined, so the limit cycle will not be accurate.

Furthermore, without the state-event handler, a third problem arises: For some systems, the state variable need to be reset after state event happens. For example, considering a bouncing ball system, the absolute velocity of the ball is reduced right after the ball hits the floor due to the energy loss in the rebounding process. It is impossible to reset the state variable using traditional ODE solvers (e.g., 'ode45.m' or 'ode15.m' etc. in MATLAB) and most existing software does not permit this.

There are three types of nonlinearities that are *unpredictable* (it is unknown when the discontinuous behavior will happen before running the system). The events caused by such nonlinearities are generated when the system state variables reach certain switching conditions. These types are:

- Switch-Type Nonlinearities
- State-Changing Nonlinearities
- Structure-Changing Nonlinearities

## 4.1 Handling Switch-Type Nonlinearities

The switch-type nonlinearities are most simple nonlinearities. The common examples include Coulomb friction and the relay, which are modeled by a change in sign of a friction term when the angular velocity passes through zero or making or breaking a connection when a voltage crosses a threshold value, respectively. Substantial errors in simulation may occur if such events can not be "captured" correctly [16], producing results that may be quite misleading when stability is marginal and which might even be confused with chaotic behavior in some cases.

The switch-type nonlinearities could be more complicated than above examples. In general a switching event can be formulated in terms of an arbitrary switching function:

$$\phi = S(x_c, t, m, u)$$

and the switching boundary is:

$$S(x_c, t, m, u) = 0$$

where $x_c$ is the CTC state vector and $m$ is the mode variable. For convenience, we define a variable $sgn$ is the sign of $S$ (i.e., $sgn = sign(S)$).

To handle such an event, the simulation software must be able to catch the points of sign changes in $S$. When such changes occur, the software should adjust the integration step that resulted in the sign change until the event has "just happened"

$(S = 0)$. It is important that there can not be any switching event in each integration step. After the software adjusted the integration step and got the point that could be an intersection of the system trajectory and the switching function, the mode variable $m$ should be reassigned to be equal to $sgn$ on the other side of the switching boundary.

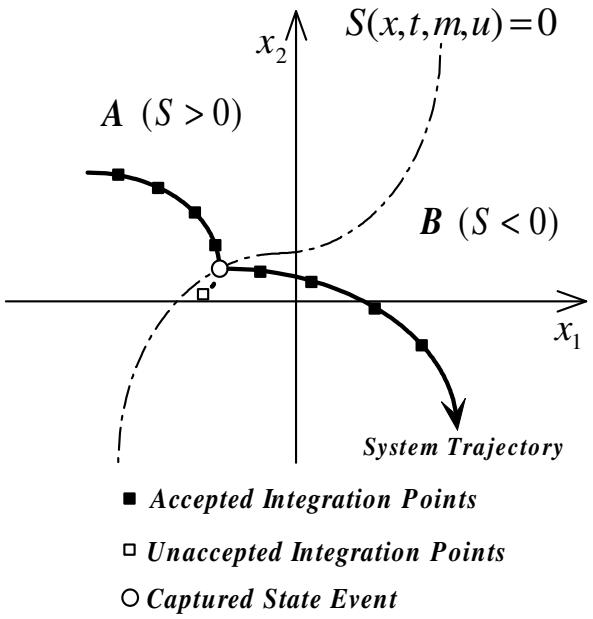


Figure 4.1: The example to show how state event handler works

The example shown in Figure 4.1 can be used to illustrate the process of handling switching type nonlinearities. In Figure 4.1, the dash dot line is the switching boundary $S(x_c, t, m, u) = 0$ and the solid line is the system trajectory. The switching boundary divides the state plane into two regions A and B. In region A, $S > 0$ and in region B, $S < 0$. We suppose the system trajectory begins in region A. The trajectory

is smooth in that region and integration is accurate and efficient. After several integration steps (represented by the solid squares), the trajectory tries to cross over the switching boundary and the Simulator gets the next integration step (represented by the empty square) where the switching function $S$ becomes negative. The Simulator watches the sign change in the switching function for each integration step before each point is accepted. When the switching function changes sign, the Simulator will discard the trial point (the empty square) due to the state event occurrence. Then, the Simulator will find the exact switching point (represented by the circle) using a root-finding algorithm. After locating the switching point, the simulation will restart with the mode variable $mode = -1$ in region B.

When a state event occurs, in order to find the exact switching point, a "zero-finding" algorithm is involved. The algorithm employed in the state event handler is the same as the algorithm used in the root-finding function "fzero" in MATLAB.

The algorithm is called "Brent's Method" [1]. Brent's method is a root-finding algorithm which combines root bracketing, interval bisection, and inverse quadratic interpolation. It is sometimes known as the van Wijngaarden-Deker-Brent method.

Brent's method uses a Lagrange interpolating polynomial of degree 2. Brent claims that this method will always converge as long as the values of the function are computable within a given region containing a root. Given three points $x_1$, $x_2$ and $x_3$, Brent's method fits $x$ as a quadratic function of $y$, then uses the interpolation formula:

$$x = \frac{[y - f(x_1)][y - f(x_2)]x_3}{[f(x_3) - f(x_1)][f(x_3) - f(x_2)]} + \frac{[y - f(x_2)][y - f(x_3)]x_1}{[f(x_1) - f(x_2)][f(x_1) - f(x_3)]} + \frac{[y - f(x_3)][y - f(x_1)]x_2}{[f(x_2) - f(x_3)][f(x_2) - f(x_1)]}$$

Subsequent root estimates are obtained by setting $y = 0$, giving

$$x = x_2 + \frac{P}{Q}$$

where

$$P = S[T(R - T)(x_3 - x_2) - (1 - R)(x_2 - x_1)]$$

$$Q = (T - 1)(R - 1)(S - 1)$$

with

$$R \equiv \frac{f(x_2)}{f(x_3)}$$

$$S \equiv \frac{f(x_2)}{f(x_1)}$$

$$T \equiv \frac{f(x_1)}{f(x_3)}$$

The zero-crossing is located using the algorithm illustrated above. The event itself will not be executed until the zero-crossing point finding process completes and an "accepted" next point on the switching surface is found.

Therefore, each integration step must always be taken with the same value of *sgn*.

In other words, the CTC model must be formulated so that it has a continuously-varying derivative during each step. The state event handler is used to instantiate any discontinuity.

## 4.2 Handling State-Changing Nonlinearities

The examples of state-changing nonlinearities include a motor that is coupled to a load through a gear train with backlash. Then there are three modes of operation: 'disengaged', 'engaged-turning-CW', 'engaged-turning-CCW'. When the mode is 'disengaged' there are two uncoupled second-order ODE sets describing the unrelated motions of the motor and the load; when they are 'engaged' we have $\theta_l = \theta_m \pm \delta$ where $\delta$ is one-half the backlash gap, and $\dot{\theta}_l = \dot{\theta_m}$. The most direct way to handle such model is to use separate models for two mechanical parts, including all torques acting on the motor and load, and add constraint equations:

$$K_e \cdot (\theta_l - \theta_m \mp \delta) = 0$$

$$K_e \cdot (\dot{\theta}_l - \dot{\theta_m}) = 0$$

where $K_e = 0$ if gears are disengaged and $K_e = 1$ if gears are engaged.

This approach can handle a wide variety of nonlinear effects, especially for the systems with mechanical components.

## 4.3 Handling Structure-Changing Nonlinearities

Besides of the two types of nonlinearities discussed above, there is a third type of discontinuity. The continuous-time dynamic equations produce derivative vector fields that are directed into the switching boundary of such type nonlinearities on both sides. This condition indicates that the trajectory being evolved cannot simply cross the boundary. Therefore, another dynamic model which governs the motion on the boundary is required. This situation represents a structural change that may be fundamentally more difficult to handle than the state-changing case.

In order to solve these type of nonlinearities, variable *mode* should be able to be set as 0 when system trajectory hits the boundary and such structure-changing event occurs at the same time. This is the most straightforward way for the software to indicate system model that the system trajectory is currently on the boundary and the dynamic model particularly for the boundary should be evolved.

When the system trajectory remains on the boundary, variable $\phi$ should be assigned as $\phi = 0$ until the trajectory can leave the boundary. The Simulator will consider the moment of $\phi$ changing from 0 to a non-zero number as a state event.

## 4.4 State Reset

Following example shows some other issues that should be concerned in the process of state event handling design. In this example [19] we have two uncoupled systems

of the form:

$$\dot{x_1} = x_2$$

$$\dot{x_2} = -sign(x_1)$$

and similarly for $x_3$, $x_4$, $\dot{x_3} = -x_4, \dot{x_4} = -sign(x_3)$ with state events defined by

$$S_1 = x_1$$

$$S_2 = x_3$$

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

$$x_c(t_e^+) = [x_1(t_e^-) \quad 0.8x_2(t_e^-) \quad x_3(t_e^-) \quad 0.8x_4(t_e^-)]^T$$

The system trajectory is shown in Figure 4.2, where the initial condition is

$$x(0) = [0.25 \quad 0 \quad -0.25 \quad 0]^T$$

This system presents two problems: first, there are two independent nonlinearities (relays) and the state events for every relay happen simultaneously if the initial conditions for $x_3$, $x_4$ are the same as those for $x_1$, $x_2$; second, the state variables need to be reset after a state event happens. The complete model for this example is listed in Appendix D.

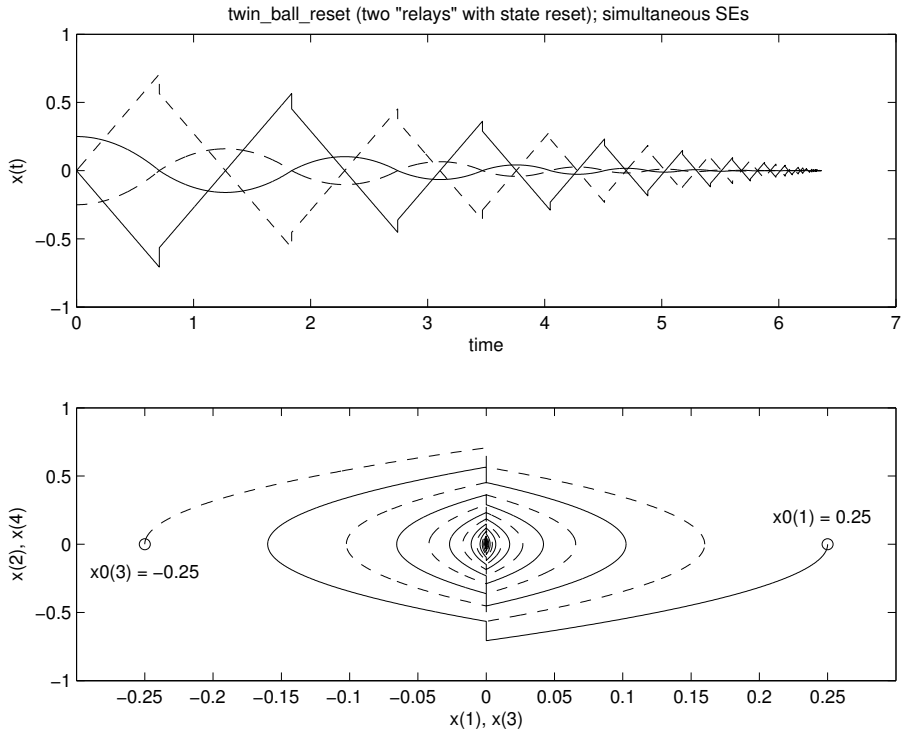


Figure 4.2: The twin relays system

To address the first problem, variables *mode* and $\phi$ should be designed as vectors, whose elements can represent each independent state event respectively. For the example discussed above, variables *mode* and $\phi$ should be two-dimensional vectors. The Simulator checks each element of $\phi$ and catches the first occurring state event or simultaneously occurring state events.

For the second problem, a reset handling part is necessary when state events occur. This part could be designed as a fundamental procedure when any state event is detected. Two major tasks are useful in this part: First, $\phi$ can be assigned as 0 if the nonlinearities are structure-changing type; second, state variable $x_c(t_e^+)$ can be

assigned to the reset value.

## 4.5 State Event Handler Flow

Figure 4.3 shows the flow of the state event handler in the Simulator. Note that this figure only shows the main procedures of the state event handler, i.e., those procedures that are not related with the state event handler are not shown in this figure, although they may take place in the flow.

According to the flow chart, the Simulator calculates the trial point $x\_trial$ and the corresponding switching function $phi\_trial$ after it gets an accepted point. Then it checks if $sign(phi) = sign(phi\_trial)$ to determine if a state event has happened. If $sign(phi) = sign(phi\_trial)$, no state event happened; the trial point is accepted and the Simulator calculates the next trial point for iteration. If $sign(phi) \neq sign(phi\_trial)$, state event(s) is(are) detected. Then the Simulator locates possible switching point(s) using the zero-finding algorithm and finds the earliest one(s). The earliest switching point(s) is (are) stored by the Simulator as a new trial point. Next, the Simulator transfers a complex *mode* to the system model to check the reset section. If $phi\_reset = 0$, the state event is an on-boundary state event. If $reset \neq [\,]$, the reset value is evaluated and stored in the output data. Finally, the Simulator updates the variable *mode*, and accepts the trial point. Then it begins to calculate the next trial point, and proceeds as above.

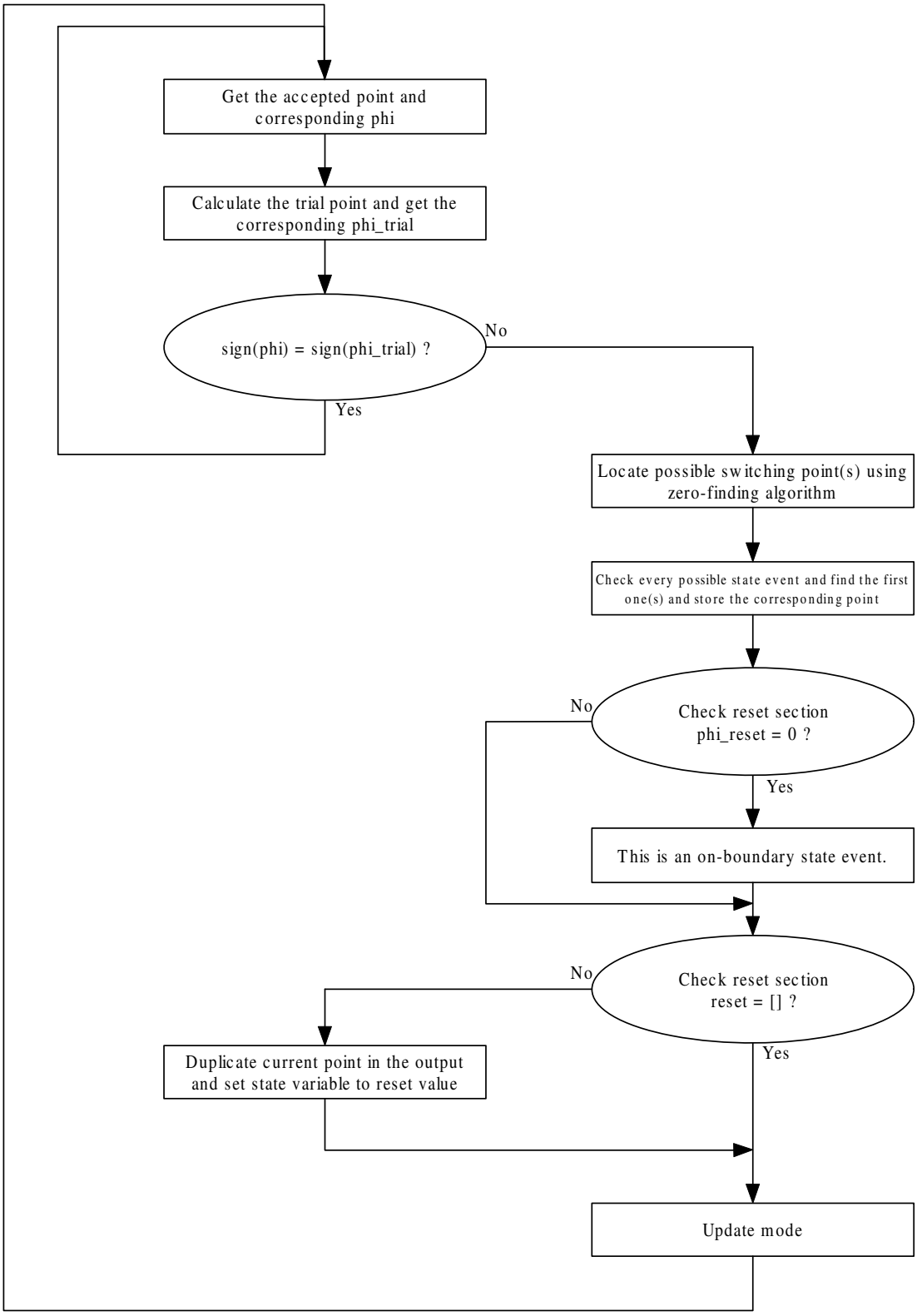


Figure 4.3: State event handler flow

# Chapter 5

# Time Event Handling

Time events are mainly generated by digital devices involved in the system. Such events happen at certain predictable times, which are generally independent of the CTC states.

The strategy to handle time events is more straightforward than that for state events. Most commercial modeling and simulation software uses a similar scheme roughly as follows:

- Before obtaining a new trial point, identify the upcoming event times for each DTC, and find $t_{next}$=time of the earliest DTC execution.
- Prepare to perform integration for CTCs using an appropriate numerical integration algorithm that could be a variable step size algorithm by checking if $t + h$ is greater than $t_{next}$; if so, set $h = t_{next} - t$.

- Perform integration with the new revised integration step to obtain the next point.
- If a state event occurs before $t_{next}$ then handle it and simulate the CTCs until $t = t_{next}$.
- Process the corresponding DTC(s).
- Restart the evolution of the CTC states.

## 5.1 Identification of DTCs and DTC State Variables

In the design of a time event handler, a fundamental problem needs to be solved. For CTCs, all state variables of every CTC can be stored together in one state vector, because all the CTCs can be handled at the same time. However, each DTC has its own sampling frequency that could be different from others. Therefore, the Simulator cannot execute all DTCs at the same time. But it is better to use only one state vector to represent all the state variables of every DTCs since it is unadvisable to set a state vector for each DTC. So, the problem is that it is necessary for the Simulator to identify which elements in the DTC state vector correspond to which DTCs.

For example, consider a system including three DTCs, the first DTC is second order, the second DTC is third order and the third DTC is first order. The column vector $x_d$ is used to represent all DTC state variables in the Simulator. The relations between the vector $xd$ and each DTC are shown in Figure 5.1. However, the Simulator cannot

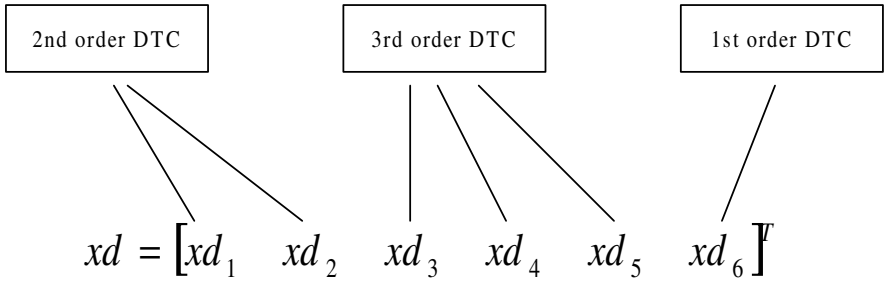


Figure 5.1: The relation between DTC state vector and every DTCs

identify the relation without certain information.

An auxiliary variable $xdh$ is employed to carry the relation information. The vector $xdh$ can be either a row vector or a column vector. The length of $xdh$ is equal to the number of DTCs included in the system. Each element in $xdh$ indicates the number of elements in the DTC state vector $xd$ corresponding to each DTC. Therefore, the following rules must be obeyed for $xdh$:

$$length(xdh) = \text{the number of DTCs}$$

$$sum(xdh) = length(xd)$$

Considering the example discussed above, $xdh$ should be:

$$xdh = [2 \quad 3 \quad 1]$$

where the number "2" indicates the first two elements in $xd$ corresponds to the first

DTC; the number "3" indicates the following three elements in $xd$ correspond to the second DTC; the number "1" indicates the following one element in $xd$ corresponds to the third DTC. With the vector $xdh$, the Simulator will be able to identify the relation between DTCs and DTC state vector $xd$ so that it can handle each DTC separately.

## 5.2 Interface Design

The interface refers to the variables that are transferred among the Simulator, the running script and the system model. In this section, we are only concerned with the variables that are exclusively used for the DTC handling.

The key to designing the interface is to determine which variables should be used in the Simulator to handle the time events. First of all, the DTC state vector $xd$ is necessary; it consists of all DTC states in a column vector form.

In the last section, the reason to involve a variable $xdh$ is introduced. The variable $xdh$ should be used in the Simulator so that it can identify the relation between DTCs and DTC state vector.

The time to execute each DTC is also critical information that the Simulator must have. A vector $te$ is employed to carry the information of the upcoming time to update every DTC. The length of $te$ equals the number of DTCs.

$$length(te) = \text{the number of DTCs}$$

So, each element in the vector *te* corresponds to the execution time of each DTC, and *te* is used by the Simulator to coordinate the time line of the entire simulation process.

The DTC state vector *xd* is transferred to the system model to be used to update DTCs when a time event happens. Another necessary variable that is passed to the system model is the previous output of each DTC. We use a column vector *yd* to represent the outputs of every DTC.

$$length(yd) = \text{the number of DTCs}$$

Each element in *yd* corresponds to each DTC in the system. The reason to involve *yd* will be explained in the next section.

The last variable that is transferred from the Simulator to the system model is the variable *ndtc*. It can be either a row vector or a column vector. The length of *ndtc* is the same as the length of *te* and *yd*.

$$length(ndtc) = \text{the number of DTCs}$$

Each element of *ndtc* corresponding to each DTC can only be assigned the value 1 or 0. It is used to inform the system model which DTC(s) should be executed when the Simulator invokes the system model. The elements that correspond to the DTCs which should be updated will be set to 1 by the Simulator before it invokes the system

model, and other elements remain 0.

Finally, the data sequence of the DTC outputs should be recorded by the Simulator. Variable *ydout* is employed to record the DTC outputs. It is a matrix variable whose size is:

$$size(ydout) = [n, m]$$

where $n$ is the number of the recorded data points, and $m$ is the number of DTCs. So, each column of *ydout* records the output of a corresponding DTC.

The variables introduced above are used in the Simulator to handle time events (DTCs). The initial condition of DTC states $xd0$ should be given to the Simulator at the beginning of a simulation. Values for $xd0$ and $xdh$ are transferred from the running script to the Simulator.

Handling DTC involves solving DTC state equations:

$$x_{d,k} = f_d(x_{d,k-1}, u_{k-1}) \tag{5.1}$$

$$y_{d,k} = g_d(x_{d,k}, u_k) \tag{5.2}$$

Equation 5.1 can also be written in the following expression:

$$x_{d,k+1} = f_d(x_{d,k}, u_k) \tag{5.3}$$

Equation 5.3 and 5.2 are used in the system model. So, in the system model, $x_{d,k}$

and $u_k$ should be given, and $x_{d,k+1}$ and $y_{d,k}$ are to be calculated. The Simulator should transfer $x_{d,k}$ to the system model, and $u_k$ should be calculated in the system model before updating DTC(s). During the simulation process, *xd* ($x_{d,k}$), *yd*, *te* and *ndtc* are transferred from the Simulator to the system model. Then, after updating, the system model returns new *xdp* ($x_{d,k+1}$), *yd* ($y_{d,k}$) and *tep*. The auxiliary vector *ndtc* is generated only by the Simulator and does not need to be updated. When the simulation process is completed, the outputs of every DTC are recorded in *ydout* that will be returned back to the running script to perform further calculations and operations defined by users in the running script. The data flow of these variables is shown in Figure 5.2.

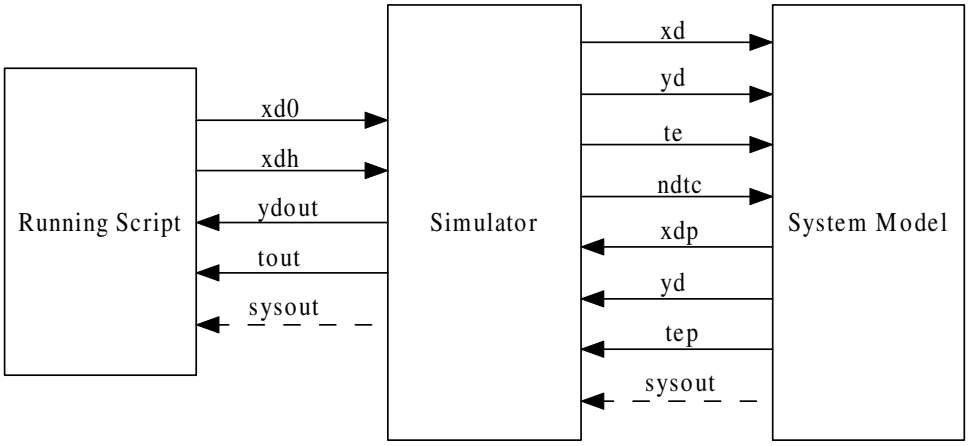


Figure 5.2: The interface for time event handler

In Figure 5.2, the output variables *yd* and *tep* refer to the new *yd* and *te* respectively after certain DTCs are updated. Variable *xdp* is the state variable at the next sampling time. So, if *xd* refers to $x_{d,k}$, *xdp* refers to $x_{d,k+1}$. In other words, *xdp* is evaluated

in the $(k-1)$th interval, to be used as $xd$ in the $k$th interval. The variable *sysout* is optional. It is the user-defined output variable.

## 5.3 Solving Digital Filters

In the last section, we mentioned that the DTC output $yd$ is necessary in the interface between the Simulator and the system model. In this section, we show that $yd$ may be needed when DTCs include digital filters.

Digital Filters are a common type of digital component. In the time domain, digital filters can be modeled by difference equations as:

$$y(k) + \sum_{i=1}^{N} a_i y(k-i) = \sum_{i=0}^{M} b_i u(k-i) \tag{5.4}$$

where $y(k)$ is the output sequence and $u(k)$ is the input sequence. Taking the z-transform of both sides of Equation 5.4, we obtain:

$$Y(z) + \sum_{i=1}^{N} a_i z^{-i} Y(z) = \sum_{i=0}^{M} b_i z^{-i} U(z)$$

Then, the transfer function is:

$$H(z) = \frac{\sum_{i=0}^{M} b_i z^{-i}}{1 + \sum_{i=1}^{N} a_i z^{-i}} \tag{5.5}$$

It also can be expressed by a state space model:

$$x(k) = Ax(k-1) + Bu(k-1) \tag{5.6}$$

$$y(k) = Cx(k) + Du(k) \tag{5.7}$$

where $x(k)$ is the state variable, $y(k)$ is the output sequence and $u(k)$ is the input sequence. As long as $b_0$ in Equation 5.5 is not zero ($b_0 \neq 0$), the parameter $D$ in Equation 5.7 will not be zero ($D \neq 0$). In this situation, the output of the digital filter, $y(k)$, depends not only on the state variable $x(k)$ but also on the digital filter input $u(k)$. In order to meet our requirement that there is an implicit zero-order hold (ZOH) at the output of each DTC, $yd$ must be remembered by the Simulator during the time interval between any two adjacent sampling instants and transferred to the system model. It is important to guarantee that $yd$ does not change until the corresponding DTC is executed.

To demonstrate this, consider an example shown in Figure 5.3:

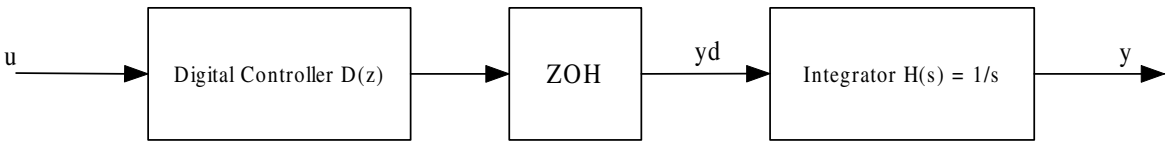


Figure 5.3: A simple system including a digital controller

where

$$D(z) = \frac{1 - 1.615z^{-1} + 0.4379z^{-2}}{1 - 0.7207z^{-1} + 0.6065z^{-2}}$$

$$H(s) = \frac{1}{s}$$

$$u = sin(t)$$

The sampling frequency of the digital controller is 2 Hz, and the input to the integrator is equal to $yd$. So in the CTC part of the system model, we have `xdot=yd;`. The simulation result is shown in Figure 5.4.

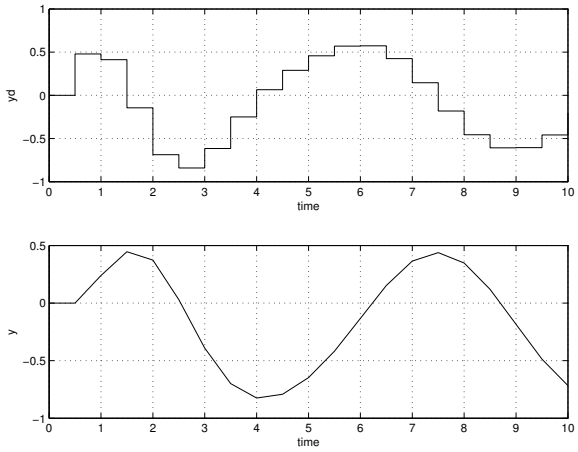


Figure 5.4: Simulation result with yd in the interface

However, if we do not have access to the variable $yd$ in the interface, i.e., the output of the ZOH is not available, it must be calculated using Equation 5.7. When the Simulator invokes the system model at a time between two adjacent sampling instants, $u(k)$ is thus replaced by $u(t)$ since $u(k)$ is not accessible in the system model. So, in the system model, we have

```
ud = sin(t);
yd = C*xd+D*ud;
```

```
xdot = yd;
```

The corresponding simulation result is shown in Figure 5.5. Note that the output of the digital controller, $yd$, is not constant between any two adjacent sampling instants. This situation conflicts with our assumption that there is an implicit zero-order hold at the output of each DTC.

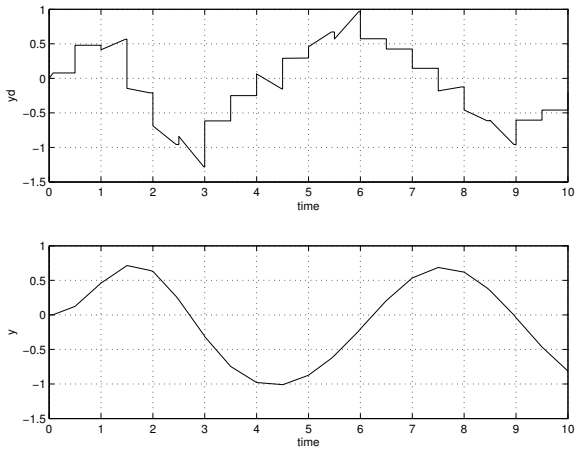


Figure 5.5: Simulation result without yd in the interface

## 5.4 Time Event Handler Flow

A brief flow diagram of the time event handler is shown in Figure 5.6. It illustrates how the time event handler works. In Figure 5.6, $N$ is the number of DTCs.

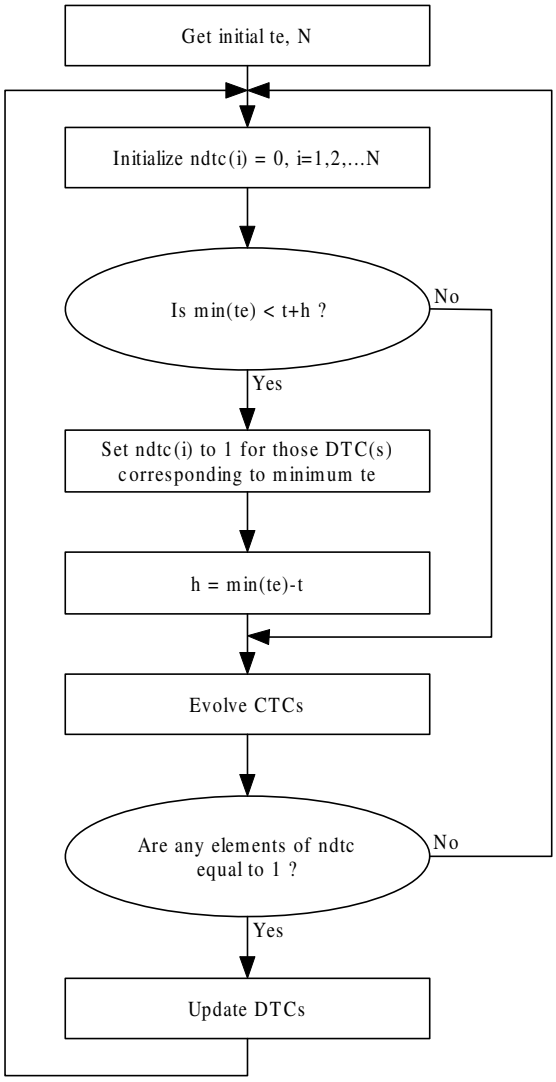


Figure 5.6: Time event handler flow

# Chapter 6

# The Simulator

## 6.1 Previous Work Analysis

In earlier research of the modeling and simulation of hybrid systems, Dr. Taylor has developed a general hybrid systems modeling language (HSML) [16]. It outlines detailed algorithms for the state event handling and the time event handling. Recently, Dr. Taylor and Mr. Kebede implemented HSML in MATLAB and created a fundamental simulation environment that include modeling schemes and a numerical integrator. However, only the state event handling algorithm is implemented in that environment, which has been posted on the web[1]. So, the environment can only handle continuous time systems. The time event handling algorithm has not been implemented. In order to improve the environment so that it can handle hybrid systems, we must implement

[1]website URL: http://www.ee.unb.ca/jtaylor/HS_software.html

the time event handling algorithm as well, and we already have the road map of it [18, 20, 21].

In order to add the mechanism for the time event handling, the structure of the previous solver (named "ode45_101.m") must be analyzed and extended. At the highest level, the structure of "ode45_101" can be expressed as a pseudo code:

```
initialization;
integration loop;
    determine the integration step size;
    calculate the trial point;
    check for any state event occurrence;
    handle the state event(s) if any occurred;
    save current data;
output data;
```

In the initialization part, some critical variables are initialized. These variables can be distinguished into three different types, according to the source of initialization. The first type of variables are initialized from the variables passed from the running script. Examples include $x0$, $t0$, and the system model name. The second type of variables are initialized directly by the Simulator. Examples include the Fehlberg coefficients $\alpha$, $\beta$, $\gamma$, the error tolerance and the initial step size etc. The third type of variables are initialized by invoking the system model; a typical example is the variable *mode*.

After the initialization for the necessary parameters, the Simulator begins the integration loop. The integration loop will not be terminated until the simulation time reaches the terminal simulation time $tfinal$, or the elapsed time exceeds the user-defined maximum elapsed time. The definitions of "Simulation Time" and "Elapsed Time" are different: "Simulation Time" refers to the time of the system that is simulated, i.e., the system running time; "Elapsed Time" refers to the real time that the computer takes when the simulation is running.

The first step in the integration loop is to determine the integration step size. Actually, this step is combined with the next step – "calculate the trial point". When the Simulator enters the integration loop, a trial point is calculated based on the initial step size. Then, the error will be evaluated. If the error exceeds the error tolerance that is defined by the user (or by default $1e-6$), the Simulator will abort the trial point and go back to redetermine a new integration step. This process will be iterated until the error corresponding to a trial point is smaller than the error tolerance.

After getting an acceptable trial point, the Simulator checks for state event existence. The switching function $\phi$ corresponding to the trial point will be calculated by invoking the system model. The $\phi$ corresponding to the trial point will be compared with the one corresponding to previous accepted point. If there is a sign change, a state event is detected. Otherwise, the trial point is accepted.

If there is a state event that is detected in the last part, the state event handler will locate the exact switching time using the root-finding algorithm that is introduced

in Chapter 4. The switching point will be the new accepted point in the system trajectory. After locating the switching point, the Simulator checks the state event handling section in the system model and update variable *mode*. If there is a reset state, the switching point will be saved in the output sequence. Then, the current accepted point will be replaced with the reset state.

The last step in the integration loop is storing the current accepted point into the output data sequence.

The integration loop is terminated when the simulation process is completed. The output variables will be returned to the running script.

## 6.2 New Solver Design and the Scope Feature

The time event handler should be inserted into the solver as in the following structure:

```
initialization;
integration loop;
    determine the step size based on the integration error;
    check time event schedule and revise step when necessary;
    calculate the trial point;
    check for state event occurrence;
    handle state event(s) if they occurred;
    handle time events if they are scheduled;
```

```
    save current data;
output data;
```

The main strategy of the time event handler has been introduced in the last chapter.

A new feature that is similar to a SIMULINK scope was added in the new solver. In some cases, users may need to access some variable in the system. The previous solver only provides users the state variables of CTCs. After adding the time event handler, the solver will also provide the output of each DTC. This is often not enough for the user's requirements. Therefore, an additional variable *sysout* is added to address the problem.

The variable *sysout* could be an $N \times M$ matrix, where $N$ is the length of the output data sequence and $M$ is the number of the user-defined output variables. In the system model, if users want to monitor $M$ variables, a $M$ dimensional vector should be assigned to the corresponding variables. Then, the Simulator will put the vector that is passed from the system model into *sysout* in each integration step. Examples will be given in Chapter 8 to illustrate how to use this feature.

## 6.3 Pseudo Code of the Simulator

The pseudo code of the Simulator is listed below.

```
%initialization
time t=t0;
```

```
maximum step size hmax=(tfinal-t)/16;
step size h=hmax/8;
CTC state variable x=x0;
DTC state variable xd=xd0;
integration step counter k=1;
store the first data into the output sequence;
call system model to get initial phi, te and xdp;
if there are user defined output variables,
    call system model to get initial output vector;
    store the output vector into sysout;
mode=sign(phi);
mdim=length(mode);

%integration loop
while (t<tfinal),
    if t+h>tfinal,
        h=tfinal-t;
    dtc_flag=0; %0 means no DTC should be updated;
    if time event(s) may happen before t+h,
        dtc_flag=1;
        find the earliest time event t_early;
        reduce step size h=t_early-t;
```

```
save current x,t,phi into xold,told,phiold;
accept_flag=0; %0 means unacceptable.
call system model to get CTC derivative xdot, integrate to t+h;
calculate error variable delta;
calculate tolerance variable tau;
if delta<=tau,
    t=t+h;
    calculate x;
    call system model with x to get new phi;
    accept_flag=1;
calculate next step size h;
if accept_flag==1,
    use zero-crossing algorithm
    if there is any state event,
        find the earliest state event, t_event and xc_tevent;
        call system model to get reset value;
        if x_reset is not empty,
            save x,t into output sequence;
            x=x_reset;
        for im=1 to mdim,
            if mode(im) changes,
                if phi_reset==0,
```

```
                    phi(im)=0;
                    mode(im)=0;
                else
                    mode(im)=sign(phi(im));
        if dtc_flag==1,
            save current DTC state vector xd into output sequence;
            for im=1 to length(xd),
                if im corresponds to a DTC state variable that should be updated,
                    xd(im)=xdp(im);
            call system model to get new xdp;
            update corresponding xdp with new value;
        store current data into output sequence;
%end
```

# Chapter 7

# Composing a System Model

## 7.1 Basic Rules

The system model must be composed under certain rules so that the Simulator can recognize and simulate the system correctly. Since the system model is a MATLAB function, the invoking format is determined by the invoking function (the Simulator). There are several formats that depend on users' requirements and the invoking format in the running script.

The standard system model invoking format is:

```
function
[xdot,phi,reset,xdp,yd,tep]=system_model(t,x,mode,xd,yd,te,ndtc)
```

Input variables:

- t: simulation time
- x: CTC state vector
- mode: mode variable used to handle state events
- xd: DTC state vector
- yd: DTC output vector
- te: DTC updating time
- ndtc: flag variable to inform which DTCs should be updated

Output variables:

- xdot: CTC state derivative vector
- phi: switching function
- reset: CTC reset state vector
- xdp: updated DTC state vector
- yd: updated DTC output vector
- tep: updated DTC execution time

All the input variables do not necessarily need to be used in the system model. However, all the output variables must be assigned no matter if they are meaningful in the system model or not, to avoid MATLAB error messages.

A normal system model format is introduced in Chapter 3 (see "system model template"). The basic framework is shown in Figure 7.1. The model can be mainly divided into two parts that are independent and must be kept separate. One part is the initialization part, and the other part is the calculation part. When the Simulator invokes the system model, only one part is executed. The input variable *mode* is used to determine which part should be executed. When $mode = [\,]$, the initialization part is activated. When $mode \neq [\,]$, the calculation part is activated.

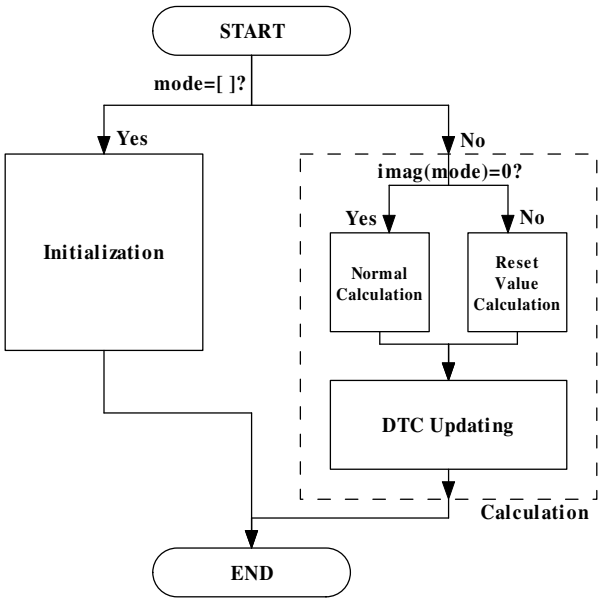


Figure 7.1: System model basic framework

In the initialization part, *xdot* and *reset* can be set to any value since they are meaningless at the beginning. Their default values are set to empty for users in the template so that users do not need to change them. The other four output variables should be assigned to the correct initial values. The variable *phi* should be assigned

to the switching function corresponding to the initial point. If the system has DTCs, $xdp$ and $yd$ should be assigned referring to the DTC state variable equations:

$$xdp = f_d(x_d, u_d)$$

$$yd = g_d(x_d, u_d)$$

where $u_d$ is the input to the DTC when $t = t0$ ($t0$ is the initial simulation time). The variable $te$ should be assigned to the first updating time(s) of the DTC(s) after simulation starts.

In the calculation part, the CTC state derivatives are calculated. The CTC derivative calculation part can also be divided into two independent sections. One section is used for the normal calculation, and the other section is used for the reset value calculation. The system model determines which section should be executed by checking if the imaginary part of the variable $mode$ is zero. If the variable $mode$ has no imaginary part ($imag(mode) = 0$), the normal calculation section is executed. If the variable $mode$ has a non-zero imaginary part ($imag(mode) \neq 0$), the reset value calculation section is activated. In the CTC derivative calculation part, the variables $xdp$ and $tep$ are useless but cannot be omitted because every output variables should be assigned. Furthermore, when DTCs are to be updated, these two variables should be initialized to the same length vectors as $xd$ and $te$ respectively in this part. So, in the system model template, two assignments $xdp = xd$, $tep = te$ are set in the CTC derivative

calculation part. Users must not change them.

If DTCs exist in the system, the DTC updating part should be added at the end of the system model. In this part, *xdp*, *yd* and *tep* can be set to the desired values. The variable *ndtc* is used by the Simulator to inform the system model which DTCs should be updated. The elements that are equal to 1 correspond to the DTCs that should be updated. Users can use a loop, as shown in the template, to check each element of *ndtc* and perform updating if $ndtc(n) = 1$.

If the system has no DTC(s), the DTC updating part can be omitted. In the initialization part, *xdp* and *yd* can be set to empty and *tep* should be set to a value bigger than the final simulation time. For example, if the simulation time is from 0 to 10 seconds, *tep* can be set to 100.

If the system has no components with modes, the variable *phi* can be set to a constant value (e.g., $phi = 1$) when it is required to be assigned.

Other system model invoking formats include the compact format and the full format:

- The compact format is used when the system has no user-defined output or DTCs. Under this circumstance, the compact invoking format is:

  ```
  function [xdot,phi,reset]=system_model(t,x,mode)
  ```

  All the variables related to DTC s are removed from the input/output interface. This invoking format is convenient for users to compose a system model if the system has no DTC, and is compatible with the previous continuous-time

Simulator [17].

- Full format is used when the system has user-defined outputs. The full invoking format is:

  ```
  function
  [xdot,phi,reset,xdp,yd,tep,sysout]=system_model(t,x,mode,xd,yd,te,ndtc)
  ```

  A user-defined output variable *sysout* is added in the output variable group. The variable *sysout* has to be assigned in the initialization part and in the calculation part.

Which format is chosen is determined by the Simulator invoking format in the running script:

- The normal Simulator invoking format is:

  ```
  [tout,xout,ydout]=ode45_sth(ypfun,t0,tfinal,x0,xd0,xdh,tol,tend,trace)
  ```

  When Simulator is invoked by above format in the running script, the standard system model invoking format should be used in the system model.

- To use the compact system model invoking format, the Simulator invoking format should be:

  ```
  [tout,xout]=ode45_sth(ypfun,t0,tfinal,x0,tol,tend,trace)
  ```

- To use the full system model invoking format, the Simulator invoking format should be:

```
[tout,xout,ydout,sysout]=
ode45_sth(ypfun,t0,tfinal,x0,xd0,xdh,tol,tend,trace)
```

The only difference among the above three formats is the number of the output variables when the running script invokes the Simulator.

## 7.2 System Model Composing Examples

### 7.2.1 Linear Open-loop System without DTC

This example is a simple linear open-loop system without DTC as shown in Figure 7.2. The input signal is $u$ and the output signal is $y$.

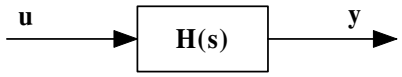


Figure 7.2: System model example 1

$$H(s) = \frac{100}{s^3 + 17s^2 + 80s + 100}$$

The input signal is a square wave.

For this system, we use the compact format to compose the system model. The system model and the running script are shown in Appendix A.

### 7.2.2 Nonlinear Open-loop System without DTC

This system example is shown in Figure 7.3. It includes a relay with hysteresis which


Figure 7.3: System model example 2

has the behavior as shown in Figure 7.4.

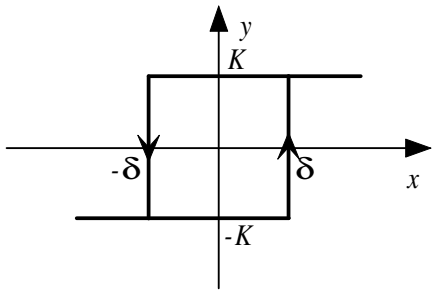


Figure 7.4: A relay with hysteresis

In Figure 7.4, $y$ is the relay output and $x$ the is the relay input.

When $y = -K$, the switching function of the relay is $\phi = x - \delta$, i.e., $\phi$ crosses zero when $x = \phi$, which is the switching condition.

When $y = K$, the switching function of the relay is $\phi = x + \delta$, by a similar argument.

In this example, the input $u$ is a square wave, $\delta = 0.1$ and $K = 1$, and

$$G(s) = \frac{10}{s + 50}$$

$$H(s) = \frac{100}{s^3 + 26s^2 + 125s + 100}$$

We also use the compact format for this system since there is no DTC in the system. The system model and the running script are shown in Appendix B. In this example, we use two sub-system-models – 'h1' and 'h2' to represent the blocks $G(s)$ and $H(s)$ respectively. It is clear but not necessary for users to build the system model in this way. The sub-system-models can be also combined in the main model.

### 7.2.3 Closed-loop System with DTC

This system is a closed-loop system with a digital controller as shown in Figure 7.5.

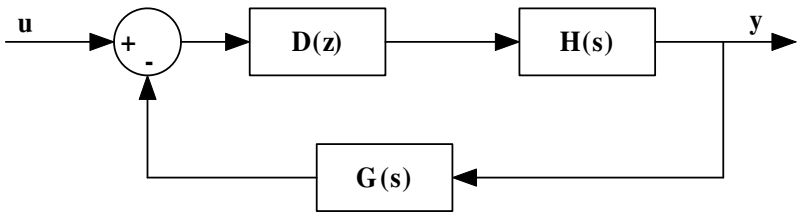


Figure 7.5: System model example 3

$$H(s) = \frac{1}{s}$$

$$G(s) = \frac{1}{s+1}$$

$$D(z) = \frac{1}{z^{-5} + 0.7z^{-4} - 0.56z^{-3} - 0.478z^{-2} + 0.02z^{-1} + 0.0762}$$

The digital controller sampling time is 0.3 second. The input $u$ is a step signal whose step time is 1 second. The system model and the running script are shown in Appendix C. In this example, sub-system-models are also employed.

### 7.2.4 System with Reset State Values

An example of a system with reset state values can be found in Section 8.1.

### 7.2.5 Hybrid Systems

An example of a close-loop hybrid system that consists a nonlinearity, CTCs and DTCs can be found in Section 8.2.

# Chapter 8

# Simulation Results and Improvement

Compared with other simulation software (we use SIMULINK as our basis of comparison), the new simulation environment has two main advantages: firstly, the generality is improved; secondly, the efficiency is improved. In this chapter, these two advantages will be illustrated and some examples will be given to support the statements.

Moreover, the new software has another new additional feature – the 'scope feature' compared with the old version [17]. An example that utilizes this feature will be given in the last section.

## 8.1 Software Generality

The software generality means the range of the system types which can be simulated using the software. The broader the range of systems that can be simulated using the software, the higher generality it has.

In SIMULINK, generally, users build system models using the blocks that are provided by SIMULINK in its library. The SIMULINK library provides a lot of blocks, each of which can represent a particular kind of system component such as a continuous-time transfer function, a discrete-time transfer function, a summation operator, a limiter, a signal generator and so on. For most basic dynamical systems, these blocks are general enough for users. When dealing with the components that are not included in the SIMULINK library such as a relay with dead band and a cubic operator etc, SIMULINK provides a special block that is called 'S-function' for users to define such components by themselves.

However, the 'S-function' approach has limitations. The following system is an example that can not be modeled using 'S-function'.

In this example, we have two uncoupled systems that have the form:

$$\dot{x_1} = x_2$$

$$\dot{x_2} = -sign(x_1)$$

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

$$S_1 = x_1$$

$$S_2 = x_3$$

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

$$x_c(t_e^+) = [x_1(t_e^-) \quad 0.8x_2(t_e^-) \quad x_3(t_e^-) \quad 0.8x_4(t_e^-)]^T$$

This example has been introduced in Section 4.4. Since the system has reset values when state events happen, SIMULINK cannot handle such a system. However, with the state event handler, our software is capable of handling such a system. The system model is shown in Appendix D. The simulation result is also shown in Section 4.4.

## 8.2 Software Efficiency

With the time event handler, the new software can handle various hybrid systems that include any kind of DTCs and logic based components. The time event handler also makes it possible to improve the simulation efficiency for those hybrid systems with nonlinear components taking the advantages of the advanced state event handler in the new software.

Consider the hybrid system shown in Figure 8.1. This is a second-order model for a

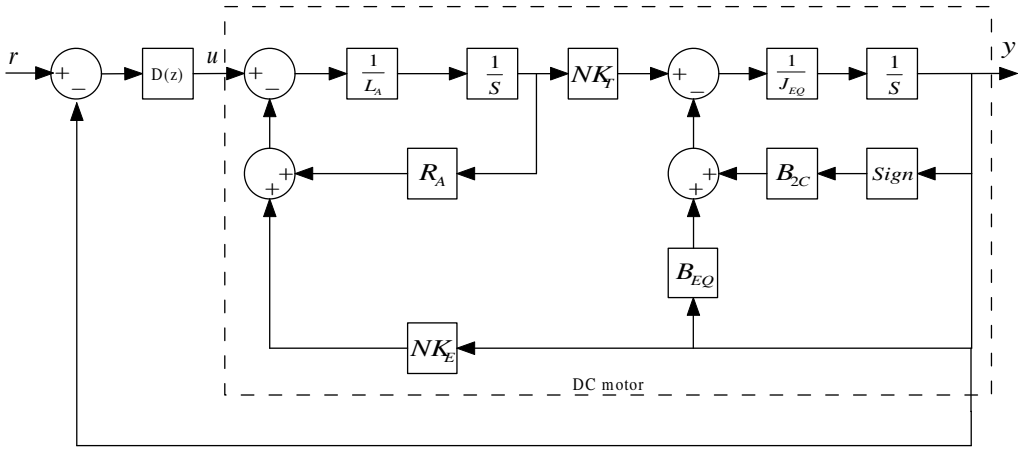


Figure 8.1: A DC motor driven by a digital controller with a negative feedback

separately-excited DC motor coupled via a gear-train to a load. The motor/load gear ratio is $N$, $K_E$ is the back emf coefficient, and $K_T$ is the torque coefficient.

If the nonlinearity $B_{2C}sign(y)$ is neglected, where y is the motor angular velocity, the motor is a second order system which has the transfer function as:

$$\frac{Y}{U} = \frac{NK_T}{J_{EQ}L_AS^2 + (J_{EQ}R_A + B_{EQ}L_A)S + B_{EQ}R_A + NK_ENK_T}$$

with the parameters:

$$K_E = 2$$

$$K_T = K_E$$

$$R_A = 0.4$$

$$L_A = 0.02$$

$$N = 10$$

$$J_e q = J_2 + N * N * J_1$$

$$B_e q = B_2 + N * N * B_1$$

$$J_1 = 5.0$$

$$B_1 = 2.0$$

$$J_2 = 70.0$$

$$B_2 = 80.0$$

The bode plot of the motor is shown in Figure 8.2.

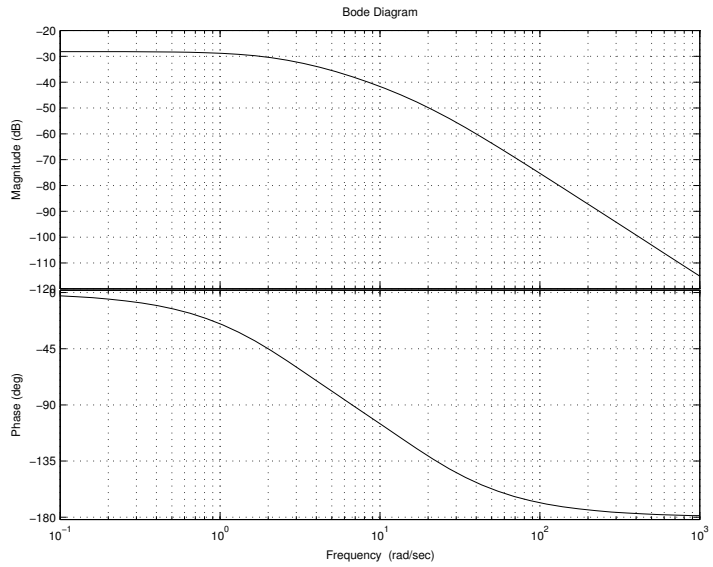


Figure 8.2: Bode plot of motor system

A digital compensator $D(z)$ in employed to improve the system performance. The digital compensator is:

$$D(z) = \frac{4z - 3.1}{z - 0.1}$$

The sampling frequency of the digital compensator is 10 Hz. The step response of the closed-loop system is shown in Figure 8.3. The digital compensator increases the system bandwidth. Therefore, the time response becomes faster and the settling time is reduced.

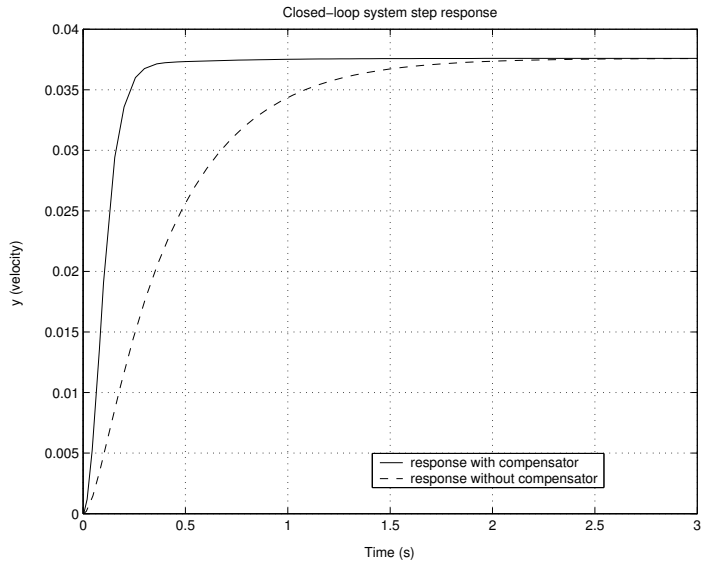


Figure 8.3: Step response

The discussion illustrated above is under the condition of $B_{2C} = 0$. The term $B_{2C}$ is the nonlinear motor friction coefficient (often called "stiction") that cannot be zero physically. When this coefficient is not zero, the nonlinear component $sign(x)$ is involved.

Assuming the friction coefficient $B_{2C} = 1500$, we will simulate the close-loop system and compare the simulation results using SIMULINK and the new software – 'ode45_sth'. The simulation interval is set to 0 to 30 seconds. The aspects that will be compared include the plot of the system output $y$ versus the time $t$ and the simulation elapsed time.

Note that the method employed for the simulation in SIMULINK is not building the system model using the blocks in the SIMULINK library, but using an 'S-function'. The reason is that Simulator 'ode45_sth' and the system model composed for it are programs in MATLAB file format. When 'ode45_sth' is running, firstly, it will be compiled and translated into the source code of C – the programming language of MATLAB and SIMULINK. This approach involves duplication of effort, as the program and the system model has to be described twice – once in MATLAB file form, and then again in C language form. The translating process takes a long time relatively in the whole running process. In SIMULINK, if the system model is built using the blocks in the library, the duplication of effort in rewriting the model in a programming language is eliminated, because the "program" is the block diagram itself [3]. However, using an 'S-function' in SIMULINK still involves the duplication of effort, as the system model also has to be described twice – once in 'S-function' form, and then again in C language form. Therefore, in order to have a fair competition, we will employ an 'S-function' to build the system model instead of using the blocks provided in the SIMULINK library when we use SIMULINK approach.

The input to the closed loop system is a sinusoid signal with a time delay $T_0 = 0.2$. It can be expressed as:

$$r = \begin{cases} 0, & t < T_0 \\ E_0 \sin[0.1\pi(t - T_0)], & t \geq T_0 \end{cases}$$

where $E_0$ is the amplitude of the sinusoid signal. We set $E_0 = 100$.

First, the SIMULINK approach is employed. The system model is built as shown in Figure 8.4. The script of the 'S-function' – 'dcmotor' is shown in Appendix E.

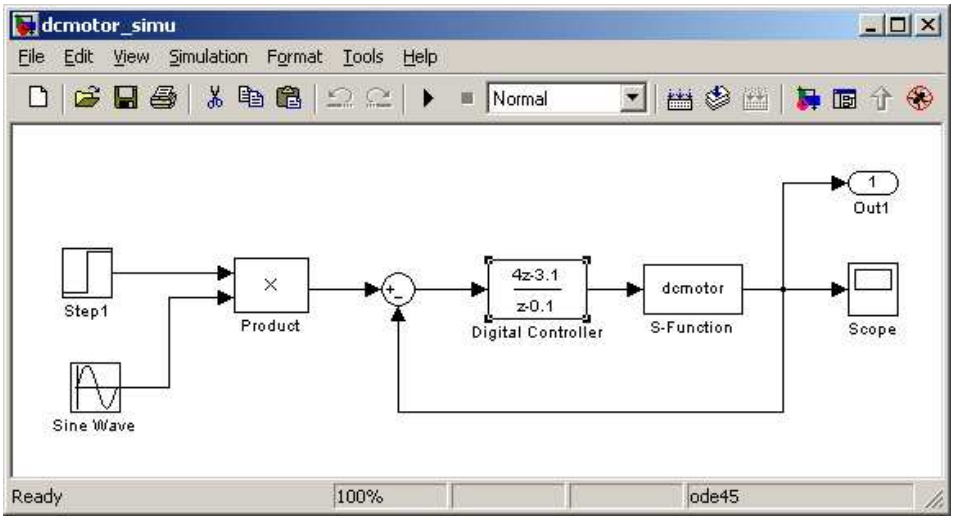


Figure 8.4: The system model built in SIMULINK

The SIMULINK simulation process takes 247 seconds, and the output data length is 336603. The simulation result of output is shown in Figure 8.5. From the bottom plot of Figure 8.5, we observe that there is a lot of chattering when the output is close to zero. Actually, the output – motor velocity should be zero when there is not enough

torque provided to the motor to overcome the stiction term. This circumstance happens when the simulation time is about $0 \sim 0.94$, $9.4 \sim 10.92$, $19.4 \sim 20.92$ and $29.4 \sim 30$ seconds. In these periods, the motor is "stuck" due to the friction. The solver of SIMULINK for this system has no state event handler. So, when the motor is supposed to be "stuck" and the electric torque provided to the motor is not zero, the solver has to make the integration step size very very small to make the error smaller than the error tolerance.

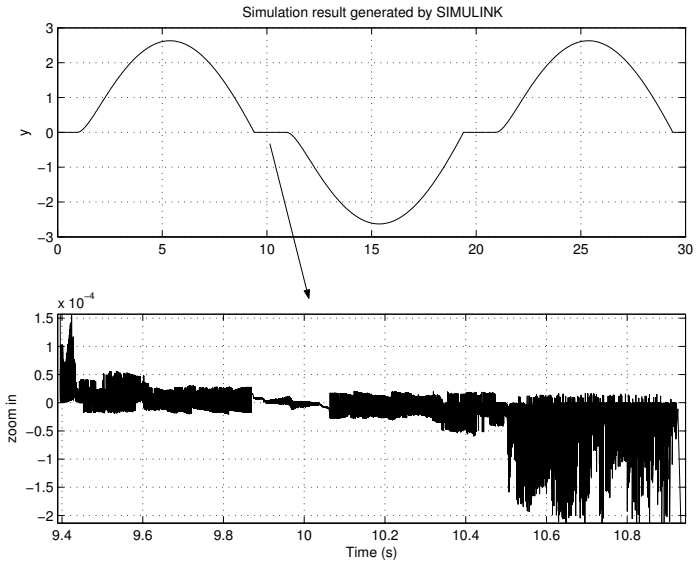


Figure 8.5: The simulation result generated by SIMULINK

Then, the new software is employed to simulate the same system. The system model and the running script are shown in Appendix F. In this model, the variable *mode* is

used to represent the motion of the motor:

$$mode = \begin{cases} 1, & y > 0 \text{ (motor rotates clockwise)} \\ 0, & y = 0 \text{ (motor stops)} \\ -1, & y < 0 \text{ (motor rotates counter-clockwise)} \end{cases}$$

The switching function is different when the motor is in different stages.

$$phi = \begin{cases} y, & mode \neq 0 \\ 0, & mode = 0 \text{ and } |T_e| < B_{2C} \\ sign(T_e), & mode = 0 \text{ and } |T_e| \geq B_{2C} \end{cases}$$

In the above equation, $T_e$ is the electric torque provided to the motor. When the motor is moving, the state event happens when the sign of output $y$ changes; When the motor is still and the electric torque is not big enough to drive the motor, there is no state event until the electric torque is big enough, at which time the state event happens.

For this system, there is an on-boundary stage when the motor stops. In the reset value setting section of the system model, *phi* should be set to zero when the motor stops.

$$phi\_reset = \begin{cases} 0, & |T_e| < B_{2C} \text{ (the motor stops)} \\ NaN, & else \text{ (other state events)} \end{cases}$$

The simulation result is shown in Figure 8.6. The simulation process takes 2.4 seconds,

and the output data length is 2077.

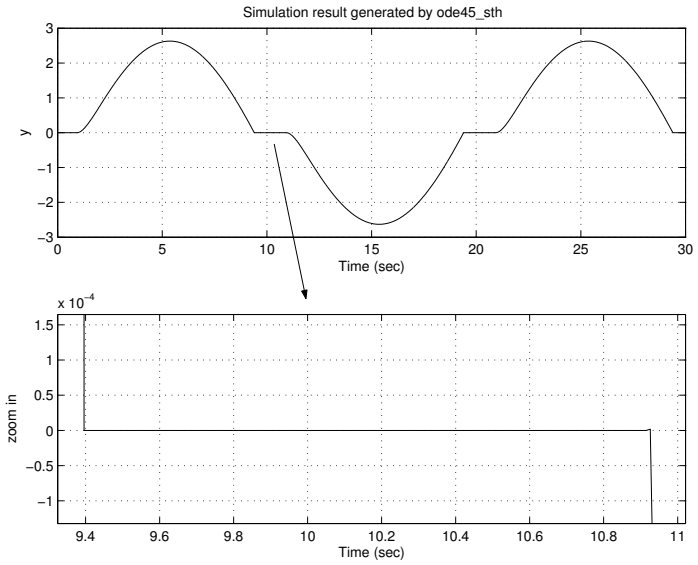


Figure 8.6: The simulation result generated by 'ode45_sth'

In the bottom plot of Figure 8.6, there is no chattering when the motor is still. Therefore, the simulating process using the new software and model is faster and much more efficient.

## 8.3 Scope Feature

The scope feature has been mentioned in Section 6.2. It offers a way for users to monitor any variable(s) in the system. It works like the 'scope' block in the SIMULINK library. Furthermore, the 'scope' block cannot show the state variables that are not the outputs of the plant, but the scope feature here is able to do this.

In the previous work [17], there are only two variables, the time variable $t$ and the

state variable $x$, which are returned from Simulator to the running script. The output of any plant in the system has to be calculated from $t$ and $x$ using the state variable equation of the plant:

$$y = g(t, x, u, m)$$

This method is feasible for those systems for which $u$ of the plant is accessible and $y$ is independent on $m$, since $m$ is unaccessible. However, in some cases, the input to the plant, $u$, is difficult, even impossible, to calculate. This leads to the problem of making the output of the plant unaccessible.

For example, consider the system shown in Figure 8.7. The 'relay' in the system has the same behavior as shown in Figure 7.4. The relay switching threshold is $\delta = \pm 0.3$. The switching off point is -0.3. The output when the relay is on is 1; when the relay is off it is -1. The system input signal is:

$$r = sin(t)$$

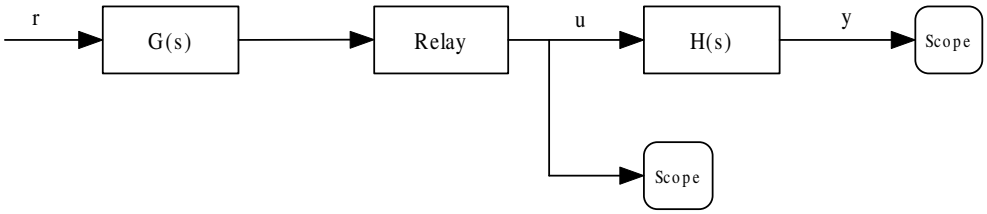


Figure 8.7: A system with scopes

$$G(s) = \frac{1}{s+1}$$

$$H(s) = \frac{s+10}{s+3}$$

Two scopes are placed at the output of the relay and the system respectively. The state variable equation of the plant $H(s)$ is:

$$\dot{x} = -3x + u$$

$$y = 7x + u$$

where $u$ is the output of the relay and $y$ is the system output – they are the two variables to be monitored. If the simulation result only has $t$ and $x$, $u$ cannot be directly obtained. The unknown $u$ also causes $y$ unaccessible.

With the scope feature of the new software, this problem can be solved. The values of $u$ and $y$ are directly recorded in the user-defined output variable *sysout*. The system model and the running script are shown in Appendix G. The simulation result plot is shown in Figure 8.8.

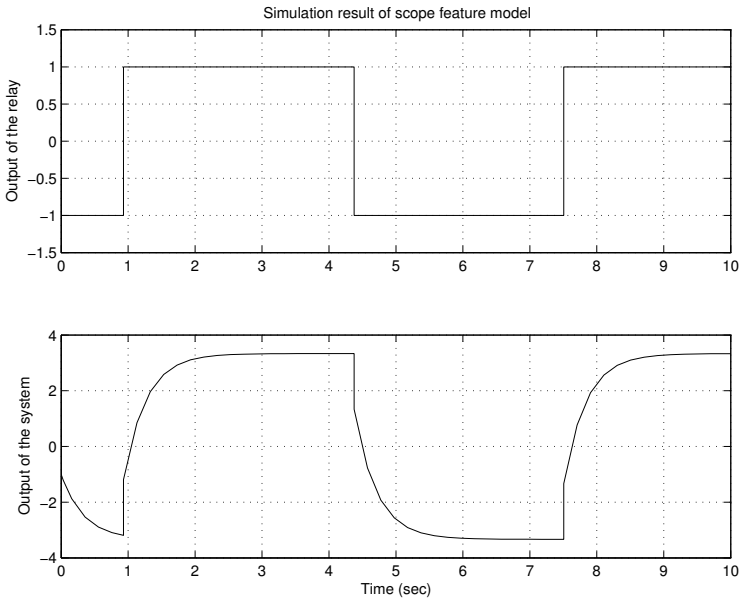


Figure 8.8: The simulation result of the system with scopes

# Chapter 9

# Conclusion and Future Work

In this project, a new simulation environment was created. Compared with the old environment, it has the following improvements: It permits

- handling DTCs with the time event handler
- observing any variables in the system easily with the scope feature incorrectly

With these improvements, the new simulation environment can simulate a wide variety of hybrid systems with greater generality than SIMULINK. With the advanced state event handler developed previously, it is more efficient for those hybrid systems which have mode-based nonlinear components. Moreover, it was designed to be as user-friendly as possible.

This environment is a handy tool for the modeling and simulation of various hybrid systems. However, there are also some disadvantages that should be improved in the

future work.

In the future work, the first step can be trying to convert the Simulator into executable form (MEX-file) within MATLAB. This approach will eliminate the compiling process for the Simulator when the user runs a simulation. Thus, it will make the simulation faster. Another way can be focused on utilizing the algorithm of this software into SIMULINK if it is possible to cooperate with MathWorks Inc. This approach can improve the software efficiency again by eliminating the translating process from MATLAB files to the source code of MATLAB. Another attempt could be cooperating with other commercial software companies to design new program compilers and graphic user interface to make the environment more efficient and user-friendly.

# Bibliography

[1] Brent, R., "Algorithms for Minimization Without Derivatives", Prentice-Hall, 1973.

[2] Cellier, F. E. "Combined continuous/discrete system simulation by use of digital computers: techniques and tools", Ph.D Dissertation, Zurich, 1979.

[3] Dabney, J.B. and Harman T.L. "Mastering Simulink 4" Prentice Hall, 2001.

[4] Kofman, E. "Quantization-Based Simulation of Hybrid Systems", Argentina, 2000.

[5] Kowalewski, S., Bauer, N., Preubig, J., Stursberg, O. and Treseler, H. "An environment for model-checking of logic control systems with hybrid dynamics", Proc. IEEE International Symposium on Computer-Aided Control System Design, Kohala Coast-Island of Hawai'i, USA, August 22-27, 1999.

[6] Linkens, D.A. and Tanyi, E.B. "Design And Implementation Of A Hybrid Modelling And Simulation Strategy For Integrated Control", Proc. IEEE International

Symposium on Computer-Aided Control System Design, Dearborn, MI, September 15-18, 1996.

[7] Moughamir, S., Zaytoon, J. and Afilal, L. "Modelling and Analysis of an Industrial Hybrid Control System", Proc. IEEE International Conference on Systems, Man, and Cybernetics, 1998.

[8] Nedialkov, N.S. and Mohrenschildt, M.V. "Rigorous simulation of hybrid dynamic systems with symbolic and interval methods", Proc. American Control Conference Anchorage, AK, May 8-10, 2002.

[9] Pizer, S.M., "Numerical Computing and Mathematical Analysis", Department of Computer Science, University of North Carolina, Chapel Hill, N.C. 1975.

[10] Rezai, M., Ito, M.R. and Lawrence, P.D. "Modeling and Simulation of Hybrid Control Systems by Global Petri Nets", Proc. IEEE International Symposium on Circuits and Systems, Seattle, Washington, USA, April 1995.

[11] Sayyar, B. "Nonlinear and hybrid modeling, simulation, and control: current technology and future views at pavil", Proc. American Control Conference, Arlington, VA, June 25-27, 2001.

[12] Stiver, J.A., Antsaklis, P.J. and Lemmon M.D. "Hybrid Control System Design Based on Natural Invariants", Proc. 34th IEEE Conference on Decision & Control, New Orleans, LA, December 1995.

[13] Sun, Y.L. and Er, M.J. "Optimal hybrid control of linear and nonlinear systems", Proc. 40th IEEE Conference on Decision and Control, Orlando, Florida, USA, December 2001.

[14] Tagawa, K., Ohta, Y. and Haneda, H. "Hybrid simulator for evaluating the performance of digital control systems", Proc. IECON'91, Japan, 1991.

[15] Taylor, J. H. "Toward a Modeling Language Standard for Hybrid Dynamical Systems", Proc. 32nd IEEE Conference on Decision and Control, pp. 2317-2322, San Antonio, TX, December 15-17, 1993.

[16] Taylor, J. H. "A Rigorous Modeling and Simulation Package for Hybrid Systems", prepared for U. S. National Science Foundation SBIR Program, Award No. III-9361232, Division of Design, Manufacturing and Industrial Innovation, 30 July 1994.

[17] Taylor, J. H. and Kebede, D. "Modeling and Simulation of Hybrid Systems", Proc. IEEE Conference on Decision and Control, New Orleans, LA, 1995.

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

[19] Taylor, J.H. "Tools for Modeling and Simulation of Hybrid Systems – A Tutorial Guide", 12 Aug 1996.

[20] Taylor, J.H. and Kebede, D. "Rigorous Hybrid Systems Simulation of an Electro-mechanical Pointing System with Discrete-time Control", Proc. American Control Conference, Albuquerque, NM, 1997.

[21] Taylor, J. H. "Rigorous Hybrid Systems Simulation with Continuous-time Discontinuities and Discrete-time Agents", Proc. 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communications and Computers, Athens, Greece, 1999.

[22] Torrisi, F.D. and Bemporad, A. "Discrete-time hybrid modeling and verification", Proc. 40th IEEE Conference on Decision and Control, Orlando, Florida, USA, December 2001.

[23] Tsybatov, V. and Vittikh, V. "A general modeling tool for hybrid systems", Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, AZ, USA, March 1994.

# Appendix A

# The system model of a linear open-loop system without DTC

**The system model:**

```
function [xdot,phi,reset]=model_1(t,x,mode)

num=100; den=[1 17 80 100];
[a,b,c,d]=tf2ss(num,den);

if isempty(mode) == true,
    xdot = [];
    phi = 1;
    reset = [];
    return;
end rf = max(abs(imag(mode)));
if rf == 0,
    u = -square(t);
    xdot = a*x+b*u;
    reset = [];
    phi = 1;
else
    xdot = [];
    phi = NaN;
    reset = [];
end
```

**the running script:**

```
clear; close all; clc;
ti = 0;
tfinal = 10;
x0 = zeros(3,1);
[t,x]=ode45_sth('model_1',ti,tfinal,x0);
num=100;
den=poly([-2 -10 -5]);
[a,b,c,d]=tf2ss(num,den);
y=c*x'-d*square(t);
plot(t,y,t,-square(t))
```

# Appendix B

# The system model of a nonlinear open-loop system without DTC

**The system model:**

```
function [xdot,phi,reset]=model_2(t,x,mode)

rf = max(abs(imag(mode)));
if isempty(mode) == true,
    u = -square(5*t);
    [temp1, temp2]=h1(x(1),u);
    if temp2 < -0.1,
        phi = -1;
    elseif temp2 > 0.1,
        phi = 1;
    else
        phi = -1;
    end
    reset = [];
    xdot = [];
    return;
end if rf == 0,
    u = -square(5*t);
    [temp1,temp2] = h1(x(1),u);
    temp5=[x(2);x(3);x(4)];
    [temp3,temp4] = h2(temp5,mode);
```

```
    xdot = [temp1(1); temp3(1); temp3(2); temp3(3)];
    if mode == 1,
        phi = temp2+0.1;
    elseif mode == -1,
        phi = temp2-0.1;
    end
    reset = [];
else
    xdot = [];
    phi = NaN;
    reset = [];
end

function [xdot,y]=h1(x,u)
num = 10;
den = [1 50];
[a,b,c,d]=tf2ss(num,den);
xdot = a*x +b*u;
y = c*x + d*u;

function [xdot,y]=h2(x,u)
num = 100;
den = [1 26 125 100];
x=x(:);
[a,b,c,d] = tf2ss(num,den);
xdot = a*x + b*u;
y = c*x +d*u;
xdot=xdot(:);
```

**the running script:**

```
clear; close all; clc;
ti = 0;
tfinal = 10;
x0 = [0;0;0;0];
[t,x]=ode45_sth('mod2',ti,tfinal,x0);
num = 100;
den = poly([-1 -5 -20]);
[a,b,c,d] = tf2ss(num,den);
```

```
c = [0 c];
y=c*x'-d*square(t);
plot(t,y,t,-square(t))
```

# Appendix C

# The system model of a close-loop system with DTC

**The system model:**

```
function
[xdot,phi,reset,xdp,yd,tep]=model_6(t,x,mode,xd,yd,te,ndtc)

Ts = 0.3;

if isempty(mode) == true,
    xdot = [];
    phi = 1;
    reset = [];
    tep = t+Ts;
    xdp = [];
    yd = [];
    return;
end

rf = max(abs(imag(mode)));
if rf == 0,
    xdot(1) = yd;
    [xdot(2), yHs] = Hs(x(2),x(1));
    xdot = xdot(:);
    phi = 1;
```

```
    reset = [];
    xdp = [];
    yd = [];
    tep = [];
else
    xdot = [];
    phi = NaN;
    reset = [];
    xdp = [];
    yd = [];
    tep = [];
end

if ndtc == true,
    if t > 1,
        u = 1-x(2);
    else
        u = -x(2);
    end
    tep = te+Ts;
    [xdp,yd] = Hz(xd,u);
end

function [xdot,y] = Hs(x,u)
%Hs = 1/(s+1)
[a,b,c,d] = tf2ss(1,[1 1]);
xdot = a*x + b*u;
y = c*x + d*u;

function [xplus,y] = Hz(x,u)
num = [1 0 0 0 0 0];
den = [1 0.7 -0.56 -0.478 0.02 0.0762];
[a,b,c,d] = tf2ss(num,den);
xplus =a*x + b*u;
y = c*x + d*u;
```

**the running script:**

```
clear; close all; clc;
```

```
x0 = zeros(2,1);
xd0 = zeros(5,1);
[t,x,ydout] = ode45_sth('model_6',0,20,x0,xd0,5);
plot(t,x(:,1));
grid on;
figure
plot(t,ydout);
grid on;
```

# Appendix D

# The system model of uncoupled relays with reset value

```
function [xdot,phi,reset] = twin_ball(t,x,mode)

if isempty(mode),
   for i=1:2,
      if x(2*i-1) == 0,
          phi(i) = x(2*i);
      else phi(i) = x(2*i-1);
      end
   end
   xdot = []; reset = [];
   return
end icall = max(abs(imag(mode))); if icall == 0,
   xdot(1) = x(2);
   xdot(2) = -mode(1);
   xdot(3) = x(4);
   xdot(4) = -mode(2);
   phi(1) = 100*x(1);
   phi(2) = 100*x(3);
   reset = [];
elseif icall == 1,
   phi = -mode;
   reset = x;
   for i=1:2,
      if abs(imag(mode(i))) == 1,
```

```
         reset(2*i-1) = 0;
         reset(2*i) = 0.8*x(2*i);
      end
   end
   xdot = [];
else
   error('bad value of mode')
end
```

# Appendix E

# The script of S-function 'dtmotor' in the system model for SIMULINK

```
function [sys,x0,str,ts] = dcmotor(t,x,u,flag,B_2C,xi)
% S-function for separately-excited DC motor coupled via
% a gear-train to a load, plus a thermal model.

%parameter initialization
J_1 = 5.0;     % kg*m^2
B_1 = 2.0;     % N*m*s/rad
% Constant flux model as in Nise
K_E = 2; % back emf coefficient, e_m = K_E*omega_m
K_T = K_E; % torque coeffic.; in SI units K_T = K_E
R_A = 0.4;    % Ohms
L_A = 0.02;   % H
% gear-train and load parameters
J_2 = 70.0;    % kg*m^2 ; 10% value in Nise...
B_2 = 80.0;    % N*m*s/rad (viscous) ; 10% value in Nise...
N = 10;        % motor/load gear ratio; omega_1 = N omega_2
% set equivalent MoI and friction coefficients
J_eq = J_2 + N*N*J_1; B_eq = B_2 + N*N*B_1;

switch flag

    case 0 % for initialization
        sys = [2,0,1,1,0,0,1];
        x0 = xi(:)';
```

```
        str = [];
        ts = [0,0];

    case {2,9} % for updating DTC or perform termination
        sys = [];

    case 1 % calculation of derivative
        xdot(1) = ( u - R_A*x(1) - K_E*N*x(2) )/L_A;
        xdot(2) = ( K_T*N*x(1) - B_eq*x(2) - B_2C*sign(x(2)))/J_eq;
        %xdot(3) = ( R_A*x(1)^2 - (x(3) - T_Amb)/R_TM )/C_TM;
        sys = xdot(:)';

    case 3 % calculation for output, set y = x(2)
        sys = x(2);

end
```

# Appendix F

# The system model composed for 'ode45_sth'

**The system model:**

```
function
[xdot,phi,reset,xdp,ydp,tep]=dcmotor_sth_model(t,x,mode,xd,yd,te,wdc)

B_2C=1500;
T = 0.1; %DTC sampling period
A=0.1; B=1; C=-2.7; D=4;
% motor parameters, Nise, Ex. 2.23 p. 92 (4th Ed.)
J_1 = 5.0;     % kg*m^2
B_1 = 2.0;     % N*m*s/rad
% Constant flux model as in Nise
K_E = 2; % back emf coefficient, e_m = K_E*omega_m
K_T = K_E; % torque coeffic.; in SI units K_T = K_E
R_A = 0.4;    % Ohms
L_A = 0.02;   % H
% gear-train and load parameters
J_2 = 70.0;    % kg*m^2 ; 10% value in Nise...
B_2 = 80.0;    % N*m*s/rad (viscous) ; 10% value in Nise...
N = 10;        % motor/load gear ratio; omega_1 = N omega_2
% set equivalent MoI and friction coefficients
J_eq = J_2 + N*N*J_1; B_eq = B_2 + N*N*B_1;
% define e_a as a step or square-wave input starting at T_0:
```

```
Te = K_T*N*x(1); T_0 = 0.2; E_0=100;
if t < T_0,
    e_a = 0;
else
    e_a = E_0*sin(0.1*pi*(t - T_0));
end
%initialization
if isempty(mode) == true,
    phi = x(2);
    xdot = [];
    reset = [];
    xdp = 0;
    ydp = 0;
    tep = T;
    return;
end

%CTC derivative calculation
rf = max(abs(imag(mode))); if rf == 0,
    xdot(1) = ( yd - R_A*x(1) - K_E*N*x(2) )/L_A;
    xdot(2) = ( K_T*N*x(1) - B_eq*x(2) - B_2C*mode)/J_eq;
    xdot = xdot(:);
    if mode == 0,
        if abs(Te) > B_2C,
            phi = sign(Te);
            xdot(2) = ( K_T*N*x(1) - B_eq*x(2) - B_2C*phi)/J_eq;
        else
            phi = 0;
            xdot(2) = 0;
        end
    else
        phi = x(2);
    end
    reset = [];
    xdp = xd; ydp = yd; tep = te;
else  % to set reset value
    xdot = [];
    if abs(Te) < B_2C,
```

```
        phi = 0;
    else
        phi = NaN;
    end
    reset = [];
    xdp = xd; ydp = yd; tep = te;
end

if wdc == 1,
    u=( e_a - x(2));
    xdp = A*xd+B*u;
    ydp = C*xd+D*u;
    tep = te+T;
end
```

**the running script:**

```
clear; close all; clc;
%script to run dcmotor model
T_final = 30;
x0 = [0;0];
xd0 = 0;
tic;
[tout,yout,ydout]=ode45_sth('dcmotor_sth_model',0,T_final,x0,xd0);
toc;
figure
subplot(211),plot(tout,yout(:,2));
grid on;
xlabel('Time (sec)');
ylabel('y');
title('Simulation result generated by ode45\_sth');
subplot(212), plot(tout,yout(:,2));
grid on;
xlabel('Time (sec)');
ylabel('Zoom in');
```

# Appendix G

# The system model composed for the system with scopes

**The system model:**

```
function
[xdot,phi,reset,xdp,yd,tep,sysout]=scopemodel(t,x,mode,xd,yd,te,ndtc)

u = sin(t); relay_threshold = 0.3;
%initialization
if isempty(mode) == true,
    phi = -1;
    xdot = [];
    reset = [];
    xdp = [];
    yd = [];
    tep = 1000;
    sysout = [-1, -1];
    return;
end
%CTC derivative calculation
rf = max(abs(imag(mode))); if rf == 0,
    xdot(1) = -x(1)+u;
    xdot(2) = -3*x(2)+mode;
    y = 7*x(2)+mode;
    if mode == -1,
```

```
        phi = x(1)-relay_threshold;
    elseif mode == 1,
        phi = x(1)+relay_threshold;
    else
        disp('mode is not right.');
    end
    reset = [];
    xdp = xd;
    yd = yd;
    tep = te;
    sysout = [mode, y];
else  % to set reset value
    xdot = [];
    phi = NaN;
    reset = [];
    xdp = xd;
    yd = yd;
    tep = te;
    sysout = [];
end
```

**the running script:**

```
clear; clc; close all;
%script to run scope_model
tfinal = 10;
x0 = [0;0];
[t,x,xd,thescope]=ode45_sth('scopemodel',0,tfinal,x0);
figure;
subplot(211),stairs(t,thescope(:,1));
title('Simulation result of scope feature model');
ylabel('Output of the relay');
axis([0 10 -1.5 1.5]); grid on;
subplot(212),plot(t,thescope(:,2)); grid on;
ylabel('Output of the system');
xlabel('Time (sec)');
```

# Vita

Candidate's full name: Jie Zhang

University attended: Nankai University, Tianjin, China

Degree obtained: Bachelor of Science and Engineering