CHAPTER 12 A DETAILED ASCEND EXAMPLE OF A DYNAMIC SIMULATION: THE MODELING OF A SIMPLE DYNAMIC TANK

the purpose for this chapter

This chapter assumes you have read Chapter 2, “A Detailed ASCEND Example for Beginners: the modeling of a vessel,” on page 5 and Chapter 3, “Preparing a model for reuse,” on page 25.

The purpose of this chapter is to be a good first step along the path to learning how to use ASCEND for dynamic simulations. We shall lead you through the steps for creating a simple model. You will also learn the standard methods that we employ for our dynamic libraries. We will present our reasons for the steps we take.

The problem

Step 1: *We would like to create a dynamic model of a simple tank.*

topics covered

Topics covered in this chapter are:

- Converting the word description to an ASCEND model.
- Solving the model.
- Creating a script to load and execute an instance of the model.
- Integrating the model.
- View Integration Results.

12.1 CONVERTING THE WORD DESCRIPTION INTO AN ASCEND MODEL

an ASCEND model is a type definition

As stated in Section 2.1, ”Converting the word description into an ASCEND model,” on page 7**,** we need to make an instance of a type and solve the instance. So we shall start by creating a tank *type* definition. We will have to create our type definition as a text file using

a text editor. (Possible text editors are Word, Framemaker, Emacs, and Notepad, pico, vi, et c. We shall discuss editors shortly.)

We need first to decide the parts to our model. In this case we know that we need the variables listed in Table 12-1. We readily fill in the first three columns in this table, and we can also fill out the fourth column if we know the units that are associated with each of the parts. To find the

Table 12-1 Variables required for model

Symbol	**Meaning**	**Typical Units**	**ASCEND variable type**
M	Moles in Tank	mol, kmol	`mole`
dM_dt	Rate of change of Moles in tank (derivative)	mol/sec, kmol/sec	`molar_rate`
input	Feed flow rate	mol/sec, kmol/sec	`molar_rate`
output	output flow rate	mol/sec, kmol/sec	`molar_rate`
Volume	Volume of liquid in the tank	m^3,ft^3	`volume`
density	molar density of tank fluid	mol/m^3,mol/ft^3	`molar_density`
dynamic	Boolean for switching between dynamic and steady state simulations	N/A	`boolean`

ASCEND variable type needed for the fourth column use the find menu on the library window and select ATOM by units. The result of this search will be all the ASCEND variable type that have the units you entered.

We would like to be able to compute the number of moles in the tank for a given volume assuming steady state (dM_dt = 0). We would also like to be able to calculate how the volume changes if we are not at steady state. The following equations describe the simple tank system.

$$\text{dM\_dt} = input - output \qquad \textbf{(12.1)}$$

$$Volume = \frac{M}{density} \qquad \textbf{(12.2)}$$

The first equation is the differential equation that relates the input and output flows to the accumulation in the tank. The second equation is the relation of the moles in the tank to the volume of liquid and should be

rearranged to avoid division. These equations are all that is need for a simple tank.

the first version of the code for tank

```
REQUIRE "ivpsystem.a4l";
REQUIRE "atoms.a4l";

MODEL tank;
    (* List of Variables *)
    dM_dt IS_A molar_rate;
    M IS_A mole;
    input IS_A molar_rate;
    output IS_A molar_rate;
    Volume IS_A volume;
    density IS_A real_constant;
    dynamic IS_A boolean;
    t IS_A time;

    (* Equations *)
    dM_dt = input - output;
    M = Volume * density;

    (* Assignment of values to Constants *)
    density :==10 {mol/m^3};

    METHODS
    METHOD check_self;
      IF (input < 1e-4 {mole/s}) THEN
       STOP {Input dried up in tank};
      END IF;
      IF (output < 1e-4 {mole/s}) THEN
       STOP {Output dried up in tank};
      END IF;
    END check_self;

    METHOD check_all;
      RUN check_self;
    END check_all;

    METHOD default_self;
      dynamic := FALSE;
      t :=0 {sec};
      dM_dt :=0 {mol/sec};
      dM_dt.lower_bound := -1e49 {mol/sec};
    END default_self;

    METHOD default_all;
```

```
  RUN default_self;
END default_all;

METHOD bound_self;
END bound_self;

METHOD bound_all;
  RUN bound_self;
END bound_all;

METHOD scale_self;
END scale_self;

METHOD scale_all;
  RUN scale_self;
END scale_all;

METHOD seqmod;
  dM_dt.fixed :=TRUE;
  M.fixed :=FALSE;
  Volume.fixed :=TRUE;
  input.fixed :=TRUE;
  output.fixed :=FALSE;
  IF dynamic THEN
   dM_dt.fixed :=FALSE;
   M.fixed :=TRUE;
   Volume.fixed :=FALSE;
   output.fixed :=TRUE;
  END IF;
END seqmod;

METHOD specify;
  input.fixed :=TRUE;
  RUN seqmod;
END specify;

METHOD set_ode;
  (* set ODE_TYPE -1=independent variable,
   0=algebraic variable, 1=state variable,
   2=derivative *)
  t.ode_type :=-1;
  dM_dt.ode_type :=2;
  M.ode_type :=1;
  (* Set ODE_ID *)
  dM_dt.ode_id :=1;
  M.ode_id :=1;
```

```
    END set_ode;

    METHOD set_obs;
      (* Set OBS_ID to any integer value greater
         than 0, the variable will be recorded
         (i.e., observed) *)
      M.obs_id :=1;
      Volume.obs_id :=2;
      input.obs_id :=3;
      output.obs_id :=4;
    END set_obs;

    METHOD values;
      Volume :=5 {m^3};
      input :=100 {mole/s};
    END values;
END tank;
```

Figure 12-1 First version of the type definition for *tank*

Our model definition has the following structure for it so far:

- MODEL statement
- list of variables we intend to use in the type definition
- equations
- METHODS
- END statement

While we have put the statements in this order, we could mix them up and intermix the middle two types of statements, even going to the extreme of defining the variables after we first use them. Once the METHODS section is started no new equations or variables can be declared. The MODEL and END statements begin and end the type definition.

There are two new methods added to a dynamic model that you would not see in a steady state model, and they are the *set_ode* and *set_obs* methods. The *set_ode* method is used to setup the model for integration. The *set_obs* method is used to tell ASCEND which variables you would like to observe in the output of the integration.

Now we need to discuss the how and why of the two new methods. The *set_ode* method is used to set up the equations and variables described in the model for integration by LSODE. In order for LSODE to be able

to integrate the model, it needs to know which variable is the independent variable — in this case t (time), which variables are the derivatives, and which are the states. The way we do this is we have to add a few extra attributes to each variable. In Section 2.1, the idea of an atom was discussed with its units, default value, bounds etc. We need to add 5 more of this type of parameter. These attributes are *ode_type, ode_id, obs_id, ode_rtol* and *ode_atol*.

This now brings us to the reason there is a system.a4l and an ivpsystem.a4l. For a steady state model the new attributes discussed above are not needed, and would take up memory and introduce confusion; therefore, they are excluded for the system library. If a dynamic simulations is to be loaded and solved, the ivpsystem library needs to be loaded instead of the system library so the extra attributes will be present with each part.

We will now go through the purpose of each of these attributes. First *ode_type* is to tell the system what type of variable it is. A value of -1 for *ode_type* means the variable is the independent variable, 0 means it is an algebraic variable (default), 1 means it is a state variable, and finally 2 means it is a derivative.

The attribute *ode_id* is used to match the state variables with their derivatives and only needs to be used if the variable is a state or derivative. In the example *M* is a state and *dM_dt* is the derivative. Therefore they both need to have the same *ode_id* so ASCEND will know that they belong together. Each state and derivative pair needs to have a different ode_id; however, it does not matter what the number is as long as it is a positive integer and no other state and derivative pair has the same number.

Next *obs_id* is used by the user to flag a variable for observation while integrating. For any integer value of *obs_id* greater then 0 the variable will be observed. The result of flagging a variable for observation is that its values will be in a data column in one of two output files. One of the files of data produced with each integration contains the values of the states and the second the values of the variables flagged for observation. The default file names are y.dat and obs.dat respectfully; however, they can be changed in the solver options general menu.

Last, but not least, are the error control attributes for LSODE: *ode_rtol* and *ode_atol*. Both of these come directly from the LSODE attributes rtol and atol which are the local relative and absolute error tolerances for the variable respectively.

There is one other thing about methods that we need to discuss before moving on and that is the *seqmod* method. If you have not already noticed, it is a little different from the other examples as it has an IF statement in it. This is an important part of the dynamic simulation. It switches the degrees of freedom depending on if we are computing an initial condition or performing an integration step. We use the boolean *dynamic* to control whether we are going to solve the model as a steady state model (*dynamic* := FALSE;) or as a dynamic model (*dynamic* := TRUE;). For the current example, we have a simple tank and, for steady state, we would like to calculate the number of moles and output flow rate for a fixed tank volume and input flow rate. Also, for the model to be at steady state, we have to fix the derivative and set it equal to zero, (*dM_dt.fixed* :=TRUE; *dM_dt* :=0 {mole/s}; The derivative is normally set to zero in the default_self method to prepare the model to solve for initial steady-state conditions.) If we then want to integrate this model for a fixed output flow (as when pumping the liquid out under flow control), we would free up the volume and fix the output flow rate. The model will then compute how the liquid volume will change with time.

In dynamic simulation, an initial value integration package, such as LSODE, repeatedly asks the model to compute the time derivatives for the state variables, given fixed values for the states. Using values for *dM_dt* computed by the model, the integration package will then update the state variable, *M*, to its new value. To accommodate this calculation, we therefore fix the state variable, *M*, and free up the derivative, *dM_dt*.

12.2 SOLVING AN ASCEND INSTANCE

We are now ready to read in and compile an instance of our tank model. We are assuming that you understand how to use the scripting window, and we will show how to go about reading, compiling, solving and integrating a dynamic model using the script in Figure 12-2.

script code

```
DELETE TYPES;
READ FILE "example.a4c";

COMPILE ex OF tank;
BROWSE ex;
RUN {ex.default_self};
RUN {ex.reset};
RUN {ex.values};
SOLVE ex WITH QRSlv;
RUN {ex.check_all};
ASSIGN {ex.dynamic} TRUE;
```

```
RUN {ex.reset};
RUN {ex.set_ode};
RUN {ex.set_obs};
# User will need to edit the next line to correct path
# to the models directory
source "$env(ASCENDDIST)/models/set_intervals.tcl";
set_int 500 10 {s};
INTEGRATE ex FROM 0 TO 50 WITH BLSODE;

ASSIGN {ex.input} 120 {mole/s};
INTEGRATE ex FROM 50 TO 499 WITH BLSODE;

# In order to view integration results for both the
# integrations the user will have to go to the solver
# window, select options, general and turn off the
# overwrite integrator logs toggle.
# (NOTE: If you were then to run a different model or this
# same simulation again it would still write to the same
# files)

# In order to see both sets of data at the same time on
# one plot you will have to merge the two sets of data in
# the file. This is done with following command.

asc_merge_data_file ascend new_obs.dat obs.dat;

# This command can also be used to convert data into a
# format that can be loaded into matlab for further work.

asc_merge_data_file matlab matlab_obs.m obs.dat;

# This command can also be used to convert data into a
# format that can be loaded into excel as a tab delimited
# text file.

asc_merge_data_file excel excel_obs.txt obs.dat;
```

Figure 12-2 Script Code.

First of all reading and compiling an instance of a dynamic model is the same as a steady state model except, as stated earlier, we must load *ivpsystem.a4l* instead of *system.a4l*. The file containing *example.a4c* (see Figure 12-1) has *REQUIRE* statements to load the right system file and the file *atoms.a4l*.

Now it is time to solve the model, and this is where things start to change. We must first solve the model for its initial conditions. We set

the boolean variable *dynamic* to *FALSE* (in the *default_self* method) and run the *reset* method to get a well-posed steady-state model. We also need to run the *values* method to set the fixed values of the initial conditions. Finally we are solve, getting as the solution the initial conditions for our model.

After solving for the initial conditions, we set things up for the dynamic simulation. We set the boolean variable *dynamic* to *TRUE* and then run the *seqmod* method to give a well-posed dynamic model. We now have to establish which variables are the independent variables, the state variables and their corresponding derivatives, and tell which variables we would like to observe; we run *set_ode* and *set_obs* methods described above.

In order for ASCEND and LSODE to know what step size and how many steps we want to observe, we must load a Tcl file that defines a new script command. The file we need to load is called *set_intervals.tcl*, and it is found in the models subdirectory of the ASCEND distribution. The command *source* comes from Tcl and is used to read and execute the a set of commands in a file. The file in this case is *set_intervals.tcl* and the commands within it setup a new script command *set_int*. Once we have loaded this file, we can use the new command *set_int*[1] to set up the number of possible steps and their maximum size. Now we are ready to integrate. The way we do this is to use the *INTEGRATE* command in the script. The syntax for these command is as follows.

Syntax for set_int

```
set_int number_of_steps step_size {units of step
size(time)};
```

Syntax for INTEGRATE

```
INTGRATE compiled_model_name FROM initial_step TO
final_step WITH BLSODE;
```

The command is set up with the initial and final step so that you can integrate for a number of steps, then make step changes, and then continue to integrate another number of steps.

12.3 VIEWING SIMULATION RESULTS

To view the simulation results, open the **ASCPLOT** window using the *Tools* menu on the **Script** window. To view a plot, first use the file menu to load the data using *Load data set*. Depending on what you

1. set_lagrangeint is also defined in *set_intervals.tcl*, and you can write other Tcl functions in this style if you want to create a customized sampling schedule.

want to look at, you can load the file containing the states or the file containing the variables you flagged for observation. Once the data file is loaded, you can double click on the file name in the top window to get a list of the variables in the file. This list will appear in the left window named *Unused variables* below where you just double clicked. As you will notice on the line below, the independent variable has already been set to time. The way we select the variables we want to plot vs. time is to highlight them from the list in the left window and, using the top arrow button, move them over to the plotted variables window on the right. We then use the *View plot file* command from the *Execute* menu to view the plot.

If we now want to plot something else, we simply highlight those variables that we do not want to plot in the plotted variables window, use the other arrow to move them back to the unused variable window and then move new variables to the plotted variables window.

If we want to change the independent variable, we select the variable we want to be the new independent variable from the list in either the unused variable window or the plotted variable window and then use the appropriate down arrow to move that variable down to become the independent variable.

Graphing options

Now that you are able to view a plot, you might want to add titles or change the axis scale, line colors, and so forth. Adding titles can be done by selecting *set titles* under the *Display menu*, a new window will open in which you will have the option to add a plot title and axis labels. To change the axis scale, line color and many other features select *see options* from the *Options menu*.

Graphing in Windows

Under MS Windows the default graph program Tkxgraph gives you full control of the options without having to go through the ASCPLOT Options menu. Tkxgraph is also available for UNIX, but xgraph does a much better job drawing dashed lines with X11 than Tkxgraph does.

If you decide you don’t like the plotting tools described above you have two more options and they are to convert the ASCEND output data files so that they can be loaded by Matlab or a spreadsheet. To convert the data files a new script command needs to be introduced and the command is *asc_merge_data_file.*

Syntax for asc_merge_data_file command

asc_merge_data_file *convert_to ouput_file_name input_file_names*

The syntax for the *asc_merge_data_file* command is as follows. First of all the *convert_to* is the format you want the data converted to. There are three options *matlab, excel* or *ascend*. The *output_file_name* is

exactly that, the name of the file in which you want the converted data to be put. The *input_file_names* is also exactly that, the file name or names that you want converted. If more than one input file is given the data is combined into one output file.

If the *matlab* option is used the output file extension should be m, if *excel* is used the extension should be txt as it is a tab delimited text file and for *ascend* the extension should be dat for use with *ASCPLOT.*

You maybe wondering what exactly is this *asc_merge_data_file* command doing. In the next three paragraphs we will give a brief explanation of each of the options.

matlab conversion

When the data is converted to be used in matlab the first thing that is done is the header of the ascend data file is placed in the output file but is commented out. This is so the user can still tell when the data was created. The next thing is does is put all the data into a matrix that has the same name as the output file with var added to the end. All variable names from the ascend data file are then converted to matlab legal names by replacing the all dots and brackets with underscores(_). The new variable names are then set equal to there corresponding column of data in the matrix. Each variable then becomes a vector. When the file is run all the data is loaded and set equal to the new variable names and can easily be plotted using matlab commands.

excel conversion

When the data is converted to be used in Excel the only thing that happens is instead of the list of variables and units being a column it is turn into rows. When the data is loaded into Excel as a tab delimited text file all the data will be in columns with the first row being the units of the data and the second being the ascend variable name. The data is then easily plotted using the Excel graphing package.

ascend conversion

This is not so much a conversion as a merge and is the origin of the command. It is only useful if there are multiple headers in a file or more than one input file is given. Multiple headers in the file occur when stopping and starting integrations with the overwrite option turned off. This conversion removes all subsequent headers that are the same as the first, whether in one file or multiple, to leave one output file with what looks like one data set for plotting. If the headers are different the data will just be combined into one file and when loaded in ASCPLOT will still look like different data sets.

12.4 PREPARING A MODEL FOR REUSE

There are four major ways to prepare a model for reuse as described in Chapter 3, “Preparing a model for reuse,” on page 25**.** All of what is said there about reusable models applies to dynamic models. However, there is one thing that we think should be repeated to make clear for dynamic models, and that is parameterizing a model.

12.4.1 PARAMETERIZING THE TANK MODEL

As stated in Section 3.3 on page 32, parameterizing a model type definition alerts a future user as to which parts of this model you deem to be the most likely to be shared. An instance of a parameterized model is then created from previously defined types.

The new thing that needs to be repeated is that the *ode_id*’s of derivative and state pairs must be different even if they are in different part of a larger model. If for instance we wanted to have two tanks in series we could parameterize the tank model and connect the two tanks together with the outlet of the first tank being the feed to the second tank. However, with the *set_ode* method, as we have currently written it, the derivative and state pairs for both tanks would have the same *ode_id*’s. Our way around this is to introduce an *ode_counter* that is used to set the *ode_id*’s and is incremented after each derivative and state pair is set. The ode counter becomes one of the model parameters and is, therefore, the same in all models. We will now give an example of this to help explain.

parameterized tank model set_ode method

```
METHOD set_ode;
   (* set ODE_TYPE -1=independent variable,
    0=algebraic variable, 1=state variable,
    2=derivative *)
   t.ode_type :=-1;
   dM_dt.ode_type :=2;
   M.ode_type :=1;
   (* Set ODE_ID *)
   dM_dt.ode_id := ode_offset;
   M.ode_id := ode_offset;
   ode_offset := ode_offset+1;
END set_ode;
```

Larger model with two tank models being used as parts. set_ode method

```
METHOD set_ode;
   RUN tank_1.set_ode;
   RUN tank_2.set_ode;
```

```
END set_ode;
```

Figure 12-3 Parameterized set_ode methods.

The parameterized tank set_ode method is almost the same as the original one we wrote except it now uses *ode_offset*, an ode_counter, to set the *ode_id*'s. It may be obvious but this is how it works. When the larger model *set_ode* is run, the *set_ode* for tank_1 is run, the *ode_id*'s are set to the current value of *ode_offset*, the counter is then incremented and *set_ode* is run for tank_2 which then gets the incremented *ode_offset* so the values are now different. You can now hopefully see that we can string as may tanks together as we like, and all the derivative and state pairs *ode_id* will be different.

This same idea can be applies to setting the observed variables. The reason this is a good idea is that the variables are placed in the output files in order of there *obs_id* value. This way we can keep all variables flagged for observation from one part of a model together.

The important thing that needs to be stressed for a dynamic system is that the time variable, dynamic boolean, and ode and obs counters must be in the parameter list. All these variable need to be the same in each model to be consistent and to make sure the model gets setup correctly when the *set_ode* method is executed.

12.5 IN CONCLUSION

We have just led you step by step through the process of creating a small dynamic ASCEND model and the basics on how to view the results.