CHAPTER 1 STARTING POINTS

Our goal:	The purpose of this chapter is to help you find out what you need to read first about ASCEND IV in order to accomplish some portion of your mathematical modeling tasks. Since there is no single “best order” to learn in for all people, we list the introductory documents and their “sound bytes” concisely, in the hope that this makes your search task less difficult. If ASCEND IV is new to you, work through the first three listed in sequence, then branch to the special topics you need most[1]. Without further ado, your goals.

1.1 PRIMAL SUBJECTS

Chapter 2	**Building and solving a small mathematical model** from a “simple” problem description of a water tank. This is basic mathematical modeling of a physical system. If you have never, ever used ASCEND IV, you should probably start here to build and solve a model.
Chapter 3	**Making any model easier to share with others** by adding basic methods, scripts, and model interfaces.
Chapter 4	**Reusing a model for plotting and case studies** with an introduction to type refinement and inheritance. Defining and executing a case study to generate data and plots which indicate how your mathematical model responds to alternative input values.
Chapter 5	**Managing modeling project files** with REQUIRE and PROVIDE. ASCEND will automatically load the other type definition files you need when working on a model if you follow some simple rules.

1. If you last used ASCEND as ASCEND III running on an HP or Apollo, ASCEND IV is new to you.

Chapter 6	**Defining a plot which gathers scattered data** from your models into a plt_plot that can be viewed from the Browser window.
howto-specify (Art,Ben, in progress)	**Defining a “square” or “well-posed” problem** when your model gets big. Writing a “specify” method is the only reliable way to go, and even this is not simple unless you plan ahead. Degrees of freedom can be tricky.
Chapter 7	**Defining new types of variables** or constants when the standard library does not have what you want.
Chapter 8	**Entering correlation equations with units** and how we support degrees Farenheit.
Chapter 9	**Defining new units of measure** based on SI or other existing units.
howto-library1 (NOTES, check methods, etc)	**Getting it right the first time.** Modeling reliably in teams requires communicating all problem aspects including the goals to be met, the mathematical problem to be solved, the solution process, and the testing criteria that define an acceptable solution. You can do all these in ASCEND IV.
Chapter 10	**Making basic models easy to use later** by adding METHODS. Defining more standard methods and your own methods so you do not have to remember how you made the model work yesterday, last week, last year, or in your last incarnation. It’s almost automatic.

1.2 ENGINEERING SUBJECTS

Chapter 11	**Defining a chemical mixture and physical property calculation**s for use in process simulation. Equilibrium thermodynamics, phases, species, and all that jazz. Adding species and correlations to the database.
howto-column1 (Art, in progress)	**Defining a steady-state distillation column** in a flowsheet using the column library that comes with ASCEND IV.
howto-reactor (Duncan, in progress)	**Defining a chemical reactor model** in a flowsheet. Not a task for the faint of heart, but probably far easier than defining a new reactor in almost any commercial simulator.
Chapter 12	**Defining a simple dynamic model (initial value problem)** and watching it respond. Water level in a tank.

howto-dynamic2 (Duncan, in progress)	**Defining a complex dynamic model** using dynamic libraries. Dynamic vapor-liquid flash tank.
howto-column2 (Duncan, in progress)	**Simulating a dynamic distillation column** in a flowsheet using ASCEND.
howto-control (Duncan, in progress)	**Controlling dynamic systems, disturbances, and all those pesky graphing tools** using the Script window and Tcl.
Chapter 13	**Writing a conditional model** where which equations apply is determined by variable values or boundary expressions.
Chapter 14 (Ben, in progress)	**Defining a dynamic model with end-point conditions** (boundary value problem) using our collocation (bvp) library.

CHAPTER 2 A DETAILED ASCEND EXAMPLE FOR BEGINNERS: THE MODELING OF A VESSEL

the purpose for this chapter

You read our propaganda about the ASCEND system in which we said it was to help technical people create hard models. We said you can tackle really large models -- 100,000 equations, compiling and solving them in minutes on a PC. We also pointed out that you can readily solve the small problems many currently solve using a spreadsheet, only once posed you can solve them inside out, upside down and backwards.

This sounded intriguing so you downloaded the system and installed it. Aside from getting the load module to transfer without error (there still are network problems), this step proved quite straight forward. You double clicked the ASCEND icon on your desktop and started it up for the first time. Four windows opened up. You panicked.

Who wouldn’t?

To use this system properly requires that you learn how to use it. If you pay the price to do so - and we hope it is not a large price, then we believe you will find the tools we have provided to help you create and debug models will pay you back handsomely.

This (Chapter 2)and the next two chapters (Chapter 3 and Chapter 4) are meant to be a good first step along the path to learning how to use ASCEND. We shall lead you through the steps for creating and testing a simple model. You will also learn how to improve this model so it may be more readily shared with others. We will present our reasons for the steps we take. We shall show you all the buttons you should push as you proceed.

We strongly suggest you put time aside and go through all three of these chapters to introduce yourself to ASCEND. It should take you about two to three hours. The second chapter is particularly important if you wish to understand our approach to good modeling practices.

the problem

Step 1: *We are going to create and test an ASCEND model to compute, the mass of the metal in the sides and ends of the thin-walled cylindrical vessel shown in Figure 2-1.*

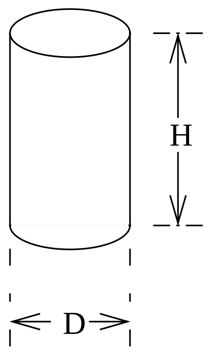

Figure 2-1 A thin-walled cylindrical vessel with flat ends

Step 2: *This model is to become a part of a library of models which others can use in the future. You must document it. You must add methods to it to make it easy for others to make it well-posed. You should probably parameterize it, and finally you must create a script which anyone can easily run that solves an example problem to illustrate its use.*

topics covered

Topics covered in this and the following two chapters are:

Chapter 2 (this chapter)

- Converting the word description to an ASCEND model.
- Loading the model into ASCEND, dealing with the error messages.
- Compiling the model.
- Browsing the model to see if it looks right
- Solving the model.
- Examining the results.
- More thoroughly testing the model.

Chapter 3

- Converting the model to a more reusable form by adding methods to it and by parameterizing it.
- Creating a script to load and execute an instance of the model.

Chapter 4

- Creating an array of models.

- Using an existing library model for plotting.
- Creating a case study using the model.

We shall introduce many of the features of the modeling language as well as the use of the interactive interface you use when compiling, debugging, solving and exploring your model. Language features include units conversion, arrays and sets.

2.1 CONVERTING THE WORD DESCRIPTION INTO AN ASCEND MODEL

an ASCEND model is a type definition

Every ASCEND model is, in fact, a type definition. To "solve a model," we make an instance of a type and solve the instance. So we shall start by creating a vessel *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. 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 2-1. We readily fill in the first three columns in this table. We shall discuss the entry in the last column in a moment.

Table 2-1 Variables required for model

Symbol	**Meaning**	**Typical Units**	**ASCEND variable type**
D	vessel diameter	m, ft	`length`
H	vessel height	m, ft	`length`
wall_thickness	wall thickness	mm, in	`length`
metal_density	metal density	kg/m^3, lbm/ft^3	`mass_density`

We will be computing the masses for the metal in the side wall and in the ends for this vessel. As this is a thin-walled vessel, we shall compute the volume of metal as the area of the walls times the wall thickness. The following equations allow us to compute the required areas

$$\text{side wall area} = \pi D H \quad \textbf{(2.1)}$$

$$\text{single end area} = \frac{\pi D^2}{4} \quad \textbf{(2.2)}$$

We should be interested in the volume of the vessel, which we compute as:

$$\text{vessel volume} = \text{end area} \times H \qquad \textbf{(2.3)}$$

We add the variables in Table 2-2 to our list.

Table 2-2 Some more variables required for vessel model

Symbol	**Meaning**	**Typical Units**	**ASCEND variable type**
side_area	area in the side wall of the vessel	m^2, ft^2	`area`
end_area	total area in the ends of the vessel	m^2, ft^2	`area`
vessel_volume	volume of the vessel	m^3, ft^3	`volume`
metal_volume	total volume of metal in walls	m^3, ft^3	`volume`
metal_mass	total mass of the metal in the walls of the ves- sel	kg, lbm	`mass`

We believe that no one should create a model of any consequence without worrying about the units for expressing the variables within it. We consider that to be a commandment handed down from somewhere on high; however, we know that others do not believe as we do. Grant us our beliefs. We have created in the ASCEND system a library of variable and constant types called

atoms.a4l.

The file type ".a4l" designates it to be an "ASCEND IV library" file. Double click on this link to see the approximately 150 different types ranging from universal constants such as π (=3.14159...) and e (=2.718...) to length, mass and angles. If we have not created one that you need, you can use this library of types to see how to construct one for yourself and add it to your file of type definitions. You will find detailed instructions for how to make your own variable type library in Chapter 7, "How to Define Variables and Scaling Values in an ASCEND Model," on page 73.

type definition library for variables and constants

ASCEND considers variable and constant types to be elementary or "atomic" to the system. These type definitions can contain only attributes for variables and constants. They cannot contain equations, for example. Thus ASCEND calls such a type definition an *atom* rather than a *model*. Figure 2-2 illustrates the definition for the type *volume*.

```
ATOM volume REFINES solver_var
   DIMENSION L^3
   DEFAULT 100.0{ft^3};
   lower_bound := 0.0{ft^3};
   upper_bound := 1e50{ft^3};
   nominal := 100.0{ft^3};
END volume;
```

Figure 2-2 A typical type definition called an atom used to define variable and constant types. Here we illustrate the type that defines volume.

The definition starts by stating that volume is a specialization of *solver_var*. The type *solver_var* refines a base type in the system known as *real* and adds several attributes to it that a nonlinear equation solver may need, such as a lower and upper bounds, a fixed flag, and so forth.

dimensions and units in ASCEND.

The type definition for volume states that volume has dimensionality of length to the power 3 (L^3) where L is one of the 10 dimensions supported by ASCEND (see “Dimensionality:” on page 161 in ASCEND Syntax document for the 10 dimensions defined within the ASCEND language).

One may express the value for a volume using any units which are consistent with the dimensionality of L^3, such as {ft^3}, {m^3}, {gal}, or even {mile^4/mm}. Setting the lower bound to 0 {ft^3} says volume must be a nonnegative number. ASCEND used the nominal value for scaling a variable of type volume when solving, here 100 ft^3.

One may change the values for the bounds, default and nominal values at any time.

We now can understand the last column in Table 2-1 and Table 2-2. For each variable or constant in the system, we have identified its type in the file *atoms.a4l*. That is, we looked in this file for the type definition that corresponded to the variable we were defining and listed that type here. This task is not as onerous as it seems. As we shall see later, we provide a tool to find for you all atom types that correspond to a particular set of units, e.g, ft^3 -- i.e., the computer will do the searching for you.

In Figure 2-3 we see the definition of one of the universal constants contained in atoms.a4l. This definition is very short; it gives the name of the type *circle_constant*, that it refines *real_constant* and that it has the value 1 {PI} where the internal conversion needed for {PI} is defined in the file defining the built-in units in ASCEND. One can add more units if desired at any time to ASCEND by defining one or more personal units files (Chapter 9 tells you how to do this).

universal constant definition

```
UNIVERSAL CONSTANT circle_constant
    REFINES real_constant :== 1{PI};
```

Figure 2-3 The type definition for circle_constant which has the value of 1 {PI} (equals 3.1415926536)

We shall in fact find this constant useful in our program, and we can either introduce a constant with this value or simply use the value 1{PI} in our program. We shall choose to do the latter.

It is time to write our first version for the model, which we do in Figure 2-4. We first list any other files containing type definitions which this model will use; here we list “atoms.a4l” following the keyword `REQUIRE`. ASCEND is sensitive to case so pay attention to where we use and do not use capital letters. Keywords are always capitalized. Often for clarification we use capital letters in a name we use for a variable or label (e.g., we use D for diameter rather than d). Note that all ASCEND statements end with a semicolon (i.e., with ;) and not at the end of a line and that blank lines have no impact. Comments are between opening and closing parenthesis/asterisk pairs, i.e., ‘*(**’ and ‘**)*’.

the first version of the code for vessel

```
REQUIRE "atoms.a4l";
MODEL vessel;
   (* variables *)
   side_area, end_area     IS_A  area;
   vessel_vol, wall_vol    IS_A  volume;
   wall_thickness, H, D    IS_A  distance;
   H_to_D_ratio            IS_A  factor;
   metal_density           IS_A  mass_density;
   metal_mass              IS_A  mass;

   (* equations *)
   FlatEnds:      end_area = 1{PI} * D^2 / 4;
   Sides:         side_area = 1{PI} * D * H;
   Cylinder:      vessel_vol = end_area * H;
   Metal_volume:  (side_area + 2 * end_area) *
                     wall_thickness = wall_vol;
   HD_definition: D * H_to_D_ratio = H;
   VesselMass:    metal_mass = metal_density * wall_vol;
END vessel;
```

Figure 2-4 First version of the type definition for *vessel.* (Available as *vesselPlain.a4c* in the ASCEND model library)

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

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

While we have put the statements in this order, we could mix up and intermix the middle two types of statements, even going to the extreme of defining the variables after we first use them. The MODEL and END statements begin and end the type definition.

You should see little that surprises you in the syntax here. However, you may have noted that we have created a definition that says absolutely nothing about how to use the variables and equations listed. There is no solution procedure buried in this type definition. In ASCEND the idea of solving is separate from saying what we intend to solve. Also note that we have not said anything about the values for any of the variables nor what we intend to calculate and what variables we intend to treat as fixed input.

2.2 EDITING, COMPILING AND BROWSING AN ASCEND MODEL

Could we compile an instance of a vessel given this definition? If there had been some arrays in our definition for which we did not say how many items were in the arrays, we could not. However, here we could compile an instance, putting aside storage space for each of the variables and somehow capturing the equations relating them.

please do not alter the **models** subdirectory

When we compile new models, we need a place to store them. One possibility would be to put them into the **models** subdirectory of the ASCEND installation (e.g., in .../ASCEND/ascend4/models/). However, you really should leave the contents of this subdirectory untouched—always. You might think of it as being read only. We count on being able to replace it totally every time you install a new version of ASCEND. Whenever we add new model libraries or corrected versions of previously existing model libraries, we put them in this subdirectory. ASCEND does nothing to enforce this rule while you run it, but please do not blame us if an upgrade wipes out changes you made in ascend4/models/; we warned you.

rather put your things into the **ascdata** subdirectory (you own it)

To avoid this problem, ASCEND also creates a subdirectory called **ascdata** that it will not touch when you install a new version of ASCEND. It will look in this subdirectory first when looking for a file to load when you have not given a full path name for finding that file. The install process for ASCEND will place **ascdata** into your home directory[1]. ASCEND tells you where it has placed this subdirectory when you start it. If you forget where it is, press the "About ASCEND IV" button on the Script window Help menu and look below the GNU ASCEND picture. It should say something like:

```
USER DATA DIRECTORY /usr0/ballan/ascdata
```

It is within the folder **ascdata** that you should place any ASCEND models you create. When running a script (which we shall talk about later), ASCEND first looks in this subdirectory for files, and then it looks in the **models** subdirectory. It stops looking when it finds the first available version of the file. For further details on this search, see Chapter 5.

create a text file containing the model definition

Next open an editor, such as Word, FrameMaker, emacs, pico, vi, vim, Notepad or Wordpad. Now type in or, better yet, cut and paste in the statements in Figure 2-4. Be very careful to match the use of capital and small letters. Do not worry about blanks between symbols but do not embed blanks within symbols. In other words, do not put a blank in the middle of the symbol *side_wall* but do not worry about putting zero or more blanks between *side_wall* and = in an equation.

When you are finished, be sure to save the file as a text file (e.g., on a PC as a .txt file). Call it vesselPlain.a4c. The ".a4c" stands for "ASCEND IV code." Editors such as Word and FrameMaker require you to use the *Save As* method to save and then to choose the file type to be *text*. Microsoft editors will append ".txt" to the file name. Remove the .txt ending off the file name -- do not let Microsoft bully you into thinking you should not -- and change it to ".a4c".

(This model is also available as *vesselPlain.a4c* in the ASCEND models library, but we suggest it would be better for you to go through

1. On Windows 95, you can identify a subdirectory to be your home directory by adding a line of the form "SET HOME=FullPathNameToSubdirectory" to the file c:\autoexec.bat. Add it without the quotes, replacing the right hand side with the full path name to the desired home directory - e.g., SET HOME=c:\mydocu~1\1998\. On UNIX and NT systems, your home directory is likely pretty obvious.

the exercise of creating your own version here. At the least copy the library file to your ASCEND space so you can play with your own version at this time.)

When you are done, you should have a text file called *vesselPlain.a4c* stored in your *ascdata* subdirectory. It should contain precisely the statements in Figure 2-4 with care having been taken to match capital and lower case letters as shown there.

start the ASCEND system. Move and resize the windows to make yourself comfortable.

Start the ASCEND system by double clicking on the ASCEND icon if you are on a PC or typing ascend at the command line if you are using a UNIX machine. Four windows[2] will appear, three smaller ones and one larger one that tells you about ASCEND. You can close this larger window by pressing its *dismiss* button. Move the three smaller ones around on your screen so they do not overlap or so they overlap very little. Resize them if you want to. You might start by putting the one called **Script** in the upper left, the one called **Library** in the upper right and the one called **Console** in the lower right. We shall assume you have placed them in these positions in the following so, even if that is not your favorite placement, it might be useful to use it for now.

The Script window shows the license and warranty information for ASCEND: ASCEND is protected by the GNU General Public License Version 2 and comes with absolutely no warranty.

note that each window by itself looks pretty nonthreatening

As you can see, each window by itself looks like a pretty normal window. Each has buttons across the top under which one will find different tools to run. Each also has one to three subwindows for displaying things. Each has a *Help* button that you can push at any time that you want to read all kinds of detailed things about the window. For the moment we will provide you with the "just in time" details here so you do not need to be sidetracked just yet by pushing these *Help* buttons.

hey, where did that window go? I want it back NOW!

If you ever lose a window, open the **Script** window and under the *Tools* button, select the window you wish to open. You cannot lose the **Script** window unless you shut down ASCEND. In the upper right of each window are Window 95/NT like buttons that iconify, enlarge and close the windows (underscore, box and X respectively). Picking X will remove the window from your screen. You get it back by going to the **Script** as described above or, as you will discover, by exporting something to it.

2. UNIX users of ASCEND will only see three windows appear. The xterm where you started ASCEND replaces the **Console** window.

I want to go to dinner (or I just panicked when I saw four windows). **How do I quit ASCEND?**

Picking the small X box in the upper right for the **Script** window is a first step in exiting ASCEND. Try it but hit the cancel button when you are asked to confirm your desire to leave. It always pays to know how (not just when) to quit. If you want to get the Script window out of the way, iconify it (pick the underscore button at the top right of the window). You will have to know how to recover an iconified window to retrieve it later - usually a simple single or double click on the icon does the trick.

saving window positions

If you like where you have placed the windows for ASCEND on your display, you can get ASCEND to remember all their locations by going to the **Script** window and selecting *Save all appearances* under the *View* button. A similar tool exists for each window for saving only its position.

start by loading and compiling using tools in the Library window

We shall start with the **Library** window in the upper right. This window provides you with the tools to load and compile files containing type definitions. You can also display the code for the different types you have loaded.

use the left mouse button unless we tell you otherwise (however, on you own explore using the right mouse button in any of the windows)

Let's load your file. Under the *File* button select the *Read types from file* tool. You select this tool by clicking on it using the left mouse button - i.e., the button you should have expected to use. A window will appear asking you to find the file you want to read into ASCEND. Navigate to where you stored *vesselPlain.a4c* (in the subdirectory *ascdata*) and select that file. If you have the wrong ending on the file (you left *.txt* or you forgot to put *.a4c* as the ending), tell the system to list all files and pick the one you want. The *.a4c* is used by the system to list only the files it thinks you might want to load, but ASCEND isn't fussy. It will attempt to load any file you pick.

Look in the Console window at the lower right, and, if the file loads without any errors being listed there, you should see

```
AscendIV% REQUIREing file “atoms.a4l”
REQUIREing file “system.a4l”
REQUIREing file “basemodel.a4l”
REQUIREing file “measures.a4l”
```

If this is what you see, you can skip past the next bit to where you should start to compile an instance. The next bit has some useful hints on how to debug your models. If you want some debugging experience, put a known error into your *vesselPlain.a4c* file and see what happens. This move will give you a reason to read the following section.

DO NOT ignore the diagnostics that might appear in the **Console** window

If the Console window in the lower right starts filling with several tens of lines of diagnostics, look to see if you included the REQUIRE statement at the beginning of your model file. Without that statement, ASCEND is missing all the definitions for the types of variables in your model, and it will go wild telling you so. It might also be choking on a Word document because you forgot to save it as a text file.

While loading the files containing these types, ASCEND will look very closely at the syntax and will give you all kinds of diagnostic messages in the Console window (lower right) if you have done something wrong. It will also at times spew out some warning messages if you have done something thought to be poor modeling style. You must heed the error messages as the file will not load if there are any. ASCEND will tell you if it did not load the file.

You should consider heeding the warnings if you get any. If you ignore them now, they may come back and haunt you later. However, there are times when we issue a warning but everything will work, and you will think we were not too clever. Our response: better modeling style can eliminate these warnings. (It’s been our system so we get to have the last word.)

how do I jump to line 100 of a file when using some of the standard editors?

The error and warning messages will contain a line number in the file where the error has occurred. This will be the line number as counted by an editor with the first line being line 1 in the file. Editors always provide you with a means to get directly to a line number in a file. Find out how to do that or you will not be too happy with debugging a large file. For example, in emacs, type a *Ctrl-c* (type the letter *c* while holding down the *Ctrl* key) followed by the letter *g*, then a line number and a carriage return. In Word and FrameMaker on the PC, type *Ctrl-g* and follow the instructions. For FrameMaker on UNIX, find the *Go to Page* tool and open it (Esc-v p or look under *View*).

You will be in the debug mode for a new system so do not expect it to be totally obvious the first few times you make an error. We have tried to use language that should be meaningful, but we may have failed or the error may be pretty subtle and not possible for us to anticipate how to describe it in your terms. (Send us a bug report if you have any good ideas on language.)

reloading a file overwrites the previous version

You can reload any file your have corrected using the *Read types from file* tool under the *File* button. It will overwrite the previous version of the file only if the file has changed since it was last loaded (pretty clever, right -- we do not reload those big files unless you make a change even if you tell us to).

displaying the code

You can display the code you have written. Select the model vessel in the right window of the Library. Then under the Display button at the top, select the tool Code. The Display window will open displaying the code for this model.

now compile as “*v*”

Okay, you have your file loaded without getting any diagnostics. You are ready to compile. In the **Library** window, look in the left window and select the file *vesselPlain.a4c*. It contains the type definition you wish to compile. You should see the type *vessel* appear in the right window. Select *vessel*. Under the *Edit* button, select *Create simulation.* A small window opens and asks you to name the simulation. Call it “*v*” -- yes, just the letter “*v*” and select “*OK*.” Short names for instances often seem to be preferable.

Look again in the **Console** window for diagnostics. If everything worked without error, you will see some statistics telling you how many models, relations and so forth you have created during the compile step.

You may see the following message in the **Console** window:

```
Found STOP statement in METHOD basemodel.a4l:239
  STOP {Error! Standard method “default_self” called but
not written in MODEL.};
  In call to METHOD default_self (depth 1) in instance v
    Line 239, File: basemodel.a4l.
```

You can safely ignore this message for now. In the next chapter, we will discuss writing methods and the meaning of this message.

and pass the instance to the **Browser**

Select *v is a vessel* in the bottom of the **Library** window. Then under the *Export* button, select *Simulation to Browser* to export *v* to the **Browser** tool set. The **Browser** window will open and contain *v*. It might be useful to enlarge this window and move it down a bit, placing it a bit to the right of the center of your computer display. (Remember you can save this positioning and sizing of the **Browser** window by going under the *View* button and picking *Save appearance*.)

examine *v* by playing with it in the **Browser**

In the left upper window of the **Browser**, you will find *v* to be the current object. Listed in the right window are all the parts of the current object. You will see the variables listed here along with an indication of their type. For example, you will find *Cylinder IS A relation* and *D IS A distance* listed, among many others. *Cylinder* is one of the equations you wrote describing the model while *D* was the diameter of the vessel.

included flags for relations

If you pick any of the parts in the right or bottom windows, it becomes the current object; its parts then show in the right window. For example, a relation has a *boolean* part (a flag that takes the value TRUE or FALSE) indicating whether or not it is to be included when ASCEND solves the equations you defined for the model.

If you wish to display the current value for this flag, pick the tool *Display Atom Values* under the *View* button. This tool toggles a switch that causes either the value or the type to show for a variable, a constant or a relation in the upper right window of the **Browser**. Try toggling it back and forth and looking at different things in the **Browser**.

Pick each of the tools under *View* and note what happens to the displaying of things in the **Browser**.

Across the bottom of the **Browser** window note the buttons you can select labeled *RV, DV* and so forth. If you have made the **Browser** window large enough, you will see to the right of these buttons the type of objects whose value you want to appear or not in the lower **Browser** window as you toggle each button. Toggle each of these buttons and see if the lower display changes. If it does not, then this type of part is not in the current object.

2.3 SOLVING AN ASCEND INSTANCE

Well, you have been patient. While there are lots of interesting tools left to explore in the **Browser**, perhaps it is time to try to solve this model. To solve *v*, make it the current object (it alone should be listed in the upper left window of the **Browser**). Then, under the *Export* button, select *to Solver*. The **Solver** window will open, along with a smaller window labeled **Eligible**. Move the **Eligible** window up a bit so it does not cover any or very little of the **Solver** window. Move the **Solver** window to the lower left and enlarge it so you can see all of its contents.

if ASCEND stops responding, hunt down one of those “nasty” windows with a “yellow lock” and close it properly

This Eligible window is one of the “nasty” ones. If it is open and you do not do something to make it happy and go away, it will stop you from doing anything else in the ASCEND system. Such windows appear with a black lock icon in a yellow field -- we shall call it a “yellow lock.” They demand you attend to them NOW. A good solution would be for such a window to stay open and on top of all the other open windows. Unfortunately we have not been able under all window managers to stop it from ducking under another window. If you ever find ASCEND unwilling to respond, iconify the other windows to get them out of the way, until you find one of these windows. On the PC you can go to the icon bar at the bottom of your screen and, by clicking on the window, bring it to the top. Then do whatever it takes to make it happy and close properly -- such as cancel it. If you are not careful here, for example, this window will hide under the **Solver** window before you are through with it.

is our problem well posed?

The **Solver** window contains the information we need to see to explain why the **Eligible** window opened in the first place. Examine the information the **Solver** displays. It tells you that *v* has 6 relations defining it and that all are equalities and included. It has no inequalities. On the right side we see there are 10 variables and all are “free.” A free variable is one for which you want the system to compute a value. Hmm, 6 equations in 10 variables. Something is wrong here. For a well-posed problem, you want 6 equations in 6 variables (i.e., square). ASCEND reports that the system is underspecified by 4. This means you need to pick four of the variables and declare them to be fixed. You will also have to pick values for these fixed variables before you can solve for the remaining 6. For such a small problem as this one, this task is not formidable. For a model with 50,000 equations and 60,000 variables, one would quit and go home. We have exposed a need here. We certainly would like ASCEND to help us here for this small problem. But we insist that it help us in major ways to make the 50,000 equation, 60,000 variable problem possible.

picking variables we are going to fix

Okay, the small help such as needed here is why the **Eligible** window opened. Let’s return to it. It lists all the variables of those not yet fixed that are eligible to be fixed and still leave us a calculation that has a chance to solve. The very fast algorithm to find eligible variables does an analysis of the structure of the equations. It cannot guarantee the resulting problem is numerically well-posed, but picking a variable it does not list as one to fix will guarantee the problem is numerically singular. Good luck on solving it if it is. We will go for coffee rather than wait for you to succeed.

So look at the list and decide what you would like to fix for your first calculation with this model. Diameter (*v.D*) seems a good choice. Now you can see why we called the instance just plain old *v*. A longer name would get tiring here. Anyway, pick *v.D*. Immediately the list reappears with *v.D* no longer on it. (ASCEND has just repeated the eligibility analysis.)

We have three more to pick. On the list are both vessel height, *v.H,* and *v.H_to_D_ratio*. We certainly cannot pick both of these. One implies the other if we know a value for *v.D*. Pick *v.H_to_D_ratio*. Note that *v.H* is no longer eligible. Good. We would be worried if it were still there.

We see *v.metal_density*. Pick it. Strange. Metal mass and volume stayed eligible. Well, okay. If we pick metal mass, wall thickness is implied, and the same is true if we were to pick metal volume. However, it seems much more natural to pick *wall_thickness* so make that the last variable picked. The **Solver** window now says this problem is square (i.e., it has 6 equations in the same number of unknowns). Table 2-3 summarizes the four variables we have elected here to fix.

Table 2-3 Variables we have fixed

variable
D
H_to_D_ratio
metal_density
wall_thickness

ASCEND partitions the problem into smaller problems for solving

Toward the bottom right of the **Solver** window, we see there are 6 “blocks.” What are blocks? ASCEND has examined the equations and, in this case, has discovered that not all the equations have to be solved simultaneously. There are 6 blocks of equations which it can solve in sequence. 6 blocks and 6 equations means that ASCEND has found a way to solve the model by solving 6 individual equations in sequence -- i.e., one at a time. That is great.

But ASCEND is going to be even smarter than this about solving in this case. If an equation is being solved by itself and if it is simple enough algebraically, ASCEND will rearrange it and solve directly for the one variable that is not yet calculated in it -- without iteration. Here all the equations are in fact that simple. This problem, with the 4 variables we selected to be fixed, can be solved entirely without iteration.

displaying the incidence matrix

Can we see what ASCEND has just discovered? It turns out we can (we would not have asked if we could not). Under the *Display* button on the **Solver**, select the *Incidence matrix* tool. A window pops open showing us the incidence of variables in the equations and display them in the order that ASCEND has found to solve them. The dark squares are incidences under the variables for which we are solving; the lighter looking X’s to the right side are incidences for the variables we have fixed. Click on the incidence in the upper left corner. ASCEND immediately identifies it for us as the end_area. It identifies the equation as the one we labeled FlatEnds. We can go back to our model and find the equation ASCEND will solve first. The other variable in this equation is in the set we fixed; pick it and discover it is D, the vessel diameter. Of course we can compute the area of the ends given the diameter. The end_area is $\pi * D^2/4$.

Play with the other incidences here. See what the other equations are and the order ASCEND will use to solve them.

Okay, we return to our task of solving. We need next to supply values for the variables we have selected to be fixed. Again, the approach we are going to take is acceptable for this small problem, but we would not want to have to do what we are about to do for a large problem. Fortunately, we really have thought about these issues and have some nice approaches that work even for extremely large problem -- like 100,000 equations.

which variables are currently fixed for this problem?

Let's see. Do you remember the variables we fixed? What if you do not? Well, we go back to the **Browser**. Be sure *v* remains the current object (it alone is in the upper left window). Under the button *Find* pick the *By type* tool. A small window opens with default information in it saying it will find for us all objects contained in the current object *v* of type *solver_var* whose fixed flags are set to *TRUE*. These are precisely the attributes for the variables we have fixed. Select *OK* and a list of the four variables we fixed earlier appears.

specifying values for the fixed variables - this approach is useful for small problems

For each variable on this list, we should supply a value. Select D in the lower window of the **Browser** using the right (the right, not the left -- make *v* the current object and do it again) mouse button. A window opens in which we input a value for *D*. Put in the value **4** in the left window and **ft** in the right. Continue by putting in the values for the variables as listed in Table 2-4. These values immediately appear in the Browser window as you enter them. If you did not fully appreciate the proper handling of dimension and units before, you just got a taste of its advantages. YOU did not have to worry about specifying these

Table 2-4 Values to use for fixed variables

variable	value	units
D	4	ft
H_to_D_ratio	3	
metal_density	5000	kg/m^3
wall_thickness	5	mm

things in consistent preselected units.

You can now solve this model. Go the **Solver** window and, under the *Execute* button, pick *Solve*. You will get a message telling you the model solved. Dismiss that message and return to the **Browser** window to examine the results. You should see the following results

```
D = 1.21922 meter
H = 3.65765 meter
H_to_D_ratio = 3
end_area = 1.16748 meter^2
metal_density = 5000 kilogram/meter^3
metal_mass = 408.62 kilogram
side_area = 14.0098 meter^2
vessel_vol = 4.27025 meter^3
wall_thickness = 0.005 meter
wall_vol = 0.0817239 meter^3
```

alter the units used for displaying values

You may wish to alter the units used to display these results. For example, you enter the diameter *D* in ft. You may wish to reassure yourself the 1.21922 meter is 4 ft. Go to the **Script** window and under the *Tools* button select *Measuring units*. The **Units** window will open. Enlarge it appropriately and then place it to the top and far right of your display.

Since length is a basic dimension in ASCEND, there is only one way to change the units for displaying length: under the *Edit* button select *Set basic units*; a cascading menu will appear, select *Length*. Another cascading menu will open with all the alternate units supported in ASCEND for length. Select *ft*. The units for all length variables will switch to *ft*. Look at the values in the **Browser** window.

The left upper window of the **Units** window contains many variable types that have composite dimensions. For example, you will find volume there. Pick it and the right window fills with all the alternative units in which you can express volume.

Play with changing the units for displaying the various variables in the vessel instance *v*.

One point - the left window displaying types having composite dimensions will display only one type for each composite dimension. If the atom types you have loaded were to include volume_scale as well as volume, then only one of the two types, volume or volume_scale, will be listed here. Changing the units to express either changes the units for both.

returning to a consistent set of units

When you are done, you may wish to return to a consistent set, such as SI. Under the *View* button are different sets; pick *SI (MKS) set*.

now we can solve the model in other ways

We can now resolve our vessel instance in any number of different ways. For example we can ask what the diameter would be if we had a volume of *250* ft^3. To accomplish this calculation, we need first to make *vessel_volume* a variable whose value we wish to fix. When we do this the model will be overspecified. ASCEND will indicate this problem to us and offer us a list of variables - including the vessel diameter *D*, one of which we will have to “unfix.” Finally we need to alter the value of *vessel_volume* to the desired value and solve. Explicit instructions to accomplish these steps are as follows.

- In the **Browser** window, make *vessel_volume* the current object (select it using the left mouse button). The right window of the **Browser** display the parts of the *vessel_volume*, among them is the *fixed* flag with a value of *FALSE*.
- (If you do not see the value for *fixed* but rather its type as a *boolean*, under the *View* button at the top, select *Display Atom Values*.)
- Pick *fixed* with the right mouse button, and, in the small window that opens, delete the value FALSE, enter the value *TRUE* and select OK.
- Now make *v* the current object by picking it in the left window of the **Browser**.
- Export *v* to the **Solver** again by selecting *to Solver* under the *Export* button. A window entitled **Overspecified** will appear listing the variables *v.D*, *v.H_to_D_ratio* and *v.vessel_volume*. Pick *v.D* and hit the *OK* button; ASCEND will reset its fixed flag to *FALSE*.
- Finally, return to the **Browser** window and select *vessel_volume* with the right mouse button. In the small window that appears type *250* in the left window, *ft^3* in the right, and hit the *OK* button.
- Under the *Execute* button in the Solver window, select Solve.

Note the **Solver** reports only 4 blocks for 6 equations. This time it has to solve some equations simultaneously. In the **Solver** window, under the *Display* button, select the *Incidence matrix* tool. You will see that the first three equations must be solved together as a single block of equations.

clearing all the *fixed* flags

For a more complicated model you may wish to start over on the process of selecting which variables are fixed. You can set the *fixed* flags for all the variables in a problem to *FALSE* all at once -- without knowing which are currently set to *TRUE*. In the **Browser** window, under the *Edit* button, select the *Run method* tool. A window will open that displays a list of default methods that are automatically attached to every model in ASCEND. One is called *ClearAll*. Pick it and hit *OK*.

All the fixed flags for the entire model will now be reset to *FALSE*. Can you think of a way to check if this is true? (Do you remember how to check which variables are currently fixed? Repeat that check and you should find no variables are on the list.)

You might now want to play by changing what you calculate and fix.

2.4 DISCUSSION

You have just completed the creation and solving of a very small model in ASCEND. In doing so, you have been exposed to some interesting issues. How can we separate the concept of the model from how we intend to solve it? How do we make a model to be well-posed -- i.e., a model involving *n* equations in *n* unknowns -- so we can solve it? How should one handle the units for the variables in a modeling system? What we have shown you here is for a small model. We still need to show you how one can make a large model well-posed, for example. You will start to understand how one can do this in the next chapter.

The next chapter is crucial for you to understand if you want to begin to understand how we approach good modeling practice. Please do continue with it. As it uses the vessel model, it would, of course, be best to continue with that chapter now.

CHAPTER 3 PREPARING A MODEL FOR REUSE

There are four major ways to prepare a model for reuse. First, you should add comments to a model. Second, you should add methods to a model definition to pass to a future user your experience in creating an instance of this type which is well-posed. Third, you should parameterize the model type definition to alert a future user as to which parts of this model you deem to be the most likely to be shared. And fourth, you should prepare a script that a future user can run to solve a sample problem involving an instance of the model. We shall consider each of these items in turn in what follows.

3.1 ADDING COMMENTS AND NOTES

In ASCEND we can create traditional comments for a model — i.e., add text to the code that aids anyone looking at the code to understand what is there. We do this by enclosing text with the delimiters (* and *). Thus the line

```
(* This is a comment *)
```

is a comment in ASCEND. Traditional comments are only visible when we display the code using the *Display code* tool in the **Library** window or when we view the code in the text editor we used to create it.

We suggest we can do more for the modeler with the concept of **Notes**, a form of "active" comments available in ASCEND. ASCEND has tools to extract notes and display them in searchable form.

notes are active comments

In Figure 3-1 we show two types of notes the modeler can add. Longer notes are set off in block style starting with the keyword NOTES and ending with END NOTES. In this model, we declare two notes in this manner: (1) to indicate who the author is and (2) to indicate the creation date for this model. Note that the notes are directed to documenting SELF which is the model itself — i.e., the vessel model as a whole object. The object one documents can be any instance in the model — any variable, equation or part. The tools for handling notes can sort on the terms enclosed in single quotes so one could, for example, isolate the *author* notes for all the models.

```
REQUIRE "atoms.a4l";
MODEL vessel;
    NOTES
   'author'          SELF {Arthur W. Westerberg}
   'creation date'   SELF {May, 1998}
    END NOTES;
   (* variables *)
   side_area      "the area of the cylindrical side wall of the vessel",
   end_area       "the area of the flat ends of the vessel"
                IS_A area;
   vessel_vol     "the volume contained within the cylindrical vessel",
   wall_vol       "the volume of the walls for the vessel"
                IS_A volume;
   wall_thickness "the thickness of all of the vessel walls",
   H              "the vessel height (of the cylindrical side walls)",
   D              "the vessel diameter"
                IS_A distance;
   H_to_D_ratio   "the ratio of vessel height to diameter"
                IS_A factor;
   metal_density  "density of the metal from which the vessel
                    is constructed"
                IS_A mass_density;
   metal_mass     "the mass of the metal in the walls of the vessel"
                IS_A mass;
   (* equations *)
FlatEnds:            end_area = 1{PI} * D^2 / 4;
Sides:               side_area = 1{PI} * D * H;
Cylinder:            vessel_vol = end_area * H;
Metal_volume:        (side_area + 2 * end_area) * wall_thickness = wall_vol;
HD_definition:       D * H_to_D_ratio = H;
VesselMass:          metal_mass = metal_density * wall_vol;
END vessel;

ADD NOTES IN vessel;
'description' SELF {This model relates the dimensions of a
           cylindrical vessel -- e.g., diameter, height and wall thickness
           to the volume of metal in the walls.  It uses a thin wall
           assumption -- i.e., that the volume of metal is the area of
           the vessel times the wall thickness.}
'purpose' SELF {to illustrate the insertion of notes into a model}
END NOTES;
```

Figure 3-1 Vessel model with Notes added (model vesselNotes.a4c)

A user may use any term desired in the single quotes. We have not decided yet what the better set of terms should be so we do not as yet suggest any. With time we expect the terms used to settle down to just a few that are repeated for all the models in a library.

there are short notes, long notes and separate notes

There are also short notes we can attach to every variable in the model. A "one liner" in double quotes just following the variable name allows the automatic annotation of variables in reports.

The last few lines of Figure 3-1 shows adding notes we write in a separate *ADD NOTES IN* object. This object can appear before or after or in a different file from the object it describes. This style of note writing is useful as it allows another person to add notes to a model without changing the code for a model. Thus it allows several different sets of notes to exist for a single model, with the choice of which to use being up to the person maintaining the model library. Finally, it allows one to eliminate the "clutter" the documentation often adds to the code.

3.2 ADDING METHODS

We would next like to pass along our experiences in getting this model to be well-posed—i.e., we would like to tell future users which variables we decided to fix and which we decided to calculate. We would also like to provide some typical values for the variables we decided to fix. ASCEND allows us to attach any number of methods to a type definition. Methods are procedural code that we can request be run through the interface while browsing a model instance. We shall include methods as described Table 3-1 to set just the right fixed flags and variable values for an instance of our vessel model to be well-posed.

The system has defaults definitions for all these methods. You already saw that to be true if you went through the process of setting all the *fixed* flags to *FALSE* in the previous chapter. In case you did not, load and compile the *vesselPlain.a4c* model in ASCEND. Export the compiled instance to the **Browser**. Then in the **Browser**, under the *Edit* button, select *Run method*. You will see a list containing these and other methods we shall be describing shortly. Select *specify* and hit the *OK* button. Then look in the Console window. A message similar to the following will appear, with all but the first line being in red to signify you should pay attention to the message:

```
Running method specify in v
Found STOP statement in METHOD
C:\PROGRAM FILES\ASCEND\ASCEND4\models\basemodel.a4l:307
```

```
STOP {Error! Standard method "specify" called but not written in MODEL.};
```

This message is telling you that you have just run the default *specify* method. We have to hand-craft every *specify* method so the default method is not appropriate. This message is alerting us to the fact that we did not yet write a special *specify* method for this model type.

Try running the *ClearAll* method. The default *ClearAll* method is always the one you will want so it does not put out a message to alert you that it is the default. d

Table 3-1 Some of the methods we require for putting a model into an ASCEND library

method	**description**
ClearAll	a method to set all the *.fixed* flags for variables in the type to *FALSE*. This puts these flags into a known standard state — i.e., all are *FALSE*. All models inherit this method from the base model and the need to rewrite it is very, very rare.
specify	a method which assumes all the fixed flags are currently *FALSE* and which then sets a suitable set of *fixed* flags to *TRUE* to make an instance of this type of model well-posed. A well-posed model is one that is square (*n* equations in *n* unknowns) and solvable.
reset	a method which first runs the ClearAll method and then the *specify* method. We include this method because it is very convenient. We only have to run one method to make any simulation well-posed, no matter how its fixed flags are currently set. All models inherit this method from the base model, as with *ClearAll*. It should only rarely have to be rewritten for a model.
values	a method to establish typical values for the variables we have fixed in an application or test model. We may also supply values for some of the variables we will be computing to aid in solving a model instance of this type. These values reflectiveness that we have tested for a simulation of this type and found to work.

writing the *specify* and *values* methods

To write the *specify* and *values* methods for our vessel model, we note that we have successfully solved the vessel model in at least two different ways above. Thus both variations are examples of being “well-posed.” We can choose which variation we shall use when creating the *specify* method for our vessel type definition. Let us choose the alternative where we fixed *vessel_volume, H_to_D_ratio, metal_density* and *wall_thicknes* and provided them with the values of *250 ft^3, 3, 5000 kg/m^3* and *5 mm* respectively to be our “standard” specification.

default methods *ClearAll* and *reset* are appropriate

As already noted, the purpose of *ClearAll* is to set all the fixed flags to *FALSE*, a well-defined state from which we can start over to set these flags as we wish them set. *Reset* simply runs *ClearAll* and then *specify* for a model. The default versions for these two methods are generally exactly what one wants so one need not write these.

Figure 3-2 illustrates our vessel model with our local versions added for *specify* and *values*. Look only at these for the moment and note that they do what we described above. We show some other methods we shall explain in a moment.

```
REQUIRE "atoms.a4l";
MODEL vessel;
    NOTES
   'author'          SELF {Arthur W. Westerberg}
   'creation date'   SELF {May, 1998}
    END NOTES;

   (* variables *)
   side_area      "the area of the cylindrical side wall of the vessel",
   end_area       "the area of the flat ends of the vessel"
                IS_A area;
   vessel_vol     "the volume contained within the cylindrical vessel",
   wall_vol       "the volume of the walls for the vessel"
                IS_A volume;
   wall_thickness "the thickness of all of the vessel walls",
   H              "the vessel height (of the cylindrical side walls)",
   D              "the vessel diameter"
                IS_A distance;
   H_to_D_ratio   "the ratio of vessel height to diameter"
                IS_A factor;
   metal_density  "density of the metal from which the vessel
                    is constructed"
                IS_A mass_density;
   metal_mass     "the mass of the metal in the walls of the vessel"
                IS_A mass;
   (* equations *)
FlatEnds:         end_area = 1{PI} * D^2 / 4;
Sides:            side_area = 1{PI} * D * H;
Cylinder:         vessel_vol = end_area * H;
Metal_volume:     (side_area + 2 * end_area) * wall_thickness = wall_vol;
HD_definition:    D * H_to_D_ratio = H;
VesselMass:       metal_mass = metal_density * wall_vol;

METHODS
METHOD specify;
```

```
    NOTES
   'purpose' SELF {to fix four variables and make the problem well-posed}
    END NOTES;
   vessel_vol.fixed        := TRUE;
   H_to_D_ratio.fixed      := TRUE;
   wall_thickness.fixed    := TRUE;
   metal_density.fixed     := TRUE;
END specify;
METHOD values;
    NOTES
   'purpose' SELF {to set the values for the fixed variables}
    END NOTES;
   H_to_D_ratio            := 2;
   vessel_vol              := 250 {ft^3};
   wall_thickness          := 5 {mm};
   metal_density           := 5000 {kg/m^3};
END values;
METHOD bound_self;
END bound_self;
METHOD scale_self;
END scale_self;
METHOD default_self;
   D                       := 1 {m};
   H                       := 1 {m};
   H_to_D_ratio            := 1;
   vessel_vol              := 1 {m^3};
   wall_thickness          := 5 {mm};
   metal_density           := 5000 {kg/m^3};
END default_self;
END vessel;

ADD NOTES IN vessel;
'description' SELF {This model relates the dimensions of a
           cylindrical vessel -- e.g., diameter, height and wall thickness
           to the volume of metal in the walls.  It uses a thin wall
           assumption -- i.e., that the volume of metal is the area of
           the vessel times the wall thickness.}
'purpose' SELF {to illustrate the insertion of notes into a model}
END NOTES;
```

Figure 3-2 Version of vessel with methods added (vesselMethods.a4c)

In Table 3-2 we describe additional methods we require before we will put a model into one of our libraries. Each of these had two versions, both of which we require. The designation *_self* is for a method to do something for all the variables and/or parts we have defined locally within the current model with an *IS_A* statement. The designation *_all* is for a method to do something for parts that are defined within an "outer" model that has an instance of this model as a part. The "outer" model is at a higher scope. It can share its parts with this model by passing them in as parameters, a topic we cover shortly in Section 3.3. Only the *_self* versions of these methods are relevant here and are in Figure 3-2.

Table 3-2 Additional methods required for model in ASCEND libraries

method	**description** (The *_self* versions of each of these methods should run the *_self* versions for the same method for all of its parts that are instances of models created with an *IS_A* statement. The *_all* version should first run the _self version of the same method and then the *_all* version for all of its parts passed in as parameters with a *WILL_BE* statement.)
default_self *default_all*	a method called automatically when any simulation is compiled to provide default values and adjust bounds for any variables which may have unsuitable defaults in their ATOM definitions. Usually the variables selected are those for which the model becomes ill-behaved if given poor initial guesses or bounds (e.g., zero).
bound_self *bound_all*	a method to update the *.upper_bound* and *.lower_bound* value for each of the variables. ASCEND solvers use these bound values to help solve the model equations.
scale_self *scale_all*	a method to update the *.nominal* value for each of the variables. ASCEND solvers will use these nominal values to rescale the variable to have a value of about one in magnitude to help solve the model equations.
check_self *check_all*	a method to check that the computations make sense. At first this method may be empty, but, with experience, one can add statements that detect answers that appear to be wrong. As ASCEND already does bounds checking, one should not check for going past bounds here. However, there could be a rule of thumb available that suggests one computed variable should be about an order of magnitude larger than another. This check could be done in this method.

adding our remaining *standard* methods to a model definition

The *bound_self* and *scale_self*, methods we have written are both empty. We anticipate no difficulties with variable scaling or bounding for this small model. Larger models can often give difficult problems in solving if the variables in them are not properly scaled and bounded; these issues must be taken very seriously for such models.

We have included the variables that define the geometry of the vessel in *defaults_self* method to avoid such things as negative initial values for *vessel_volume*. The compiler for ASCEND runs this method as soon as the model is compiled into an instance so the variables mentioned here start with their default values.

using methods when solving

Exit ASCEND and repeat all the steps above to edit, load and compile this new vessel type definition. Then proceed as follows.

- In the *Browser* window, examine the values for those variables mentioned in the *default_self* method. Note they already have their default values.
- To place the new instance *v* in a solvable state, go to the **Browser** window. Pick the command *Run method* under the *Edit* button. Select first the method *values* and hit *OK*.
- Repeat the last step but this time select the method *reset*.

In the **Browser**, examine the values for the variables listed in the method *values* in Figure 3-2. They should be set to those stated (remember you can alter the units ASCEND uses to report the values by using the tools in the **Units** window).Also examine the *fixed* flags for these variables; they should all be *TRUE* (remember that you can find which variables are fixed all at once by using the *By type* command under the *Find* button).

- Finally export *v* to the **Solver**. The **Eligible** window should NOT appear; rather that **Solver** should report the model to be *square*.
- Solve by selecting *Solve* under the *Execute* button.

The inclusion of methods has made the process of making this model much easier to get well-posed. This approach is the one that works for really large, complex models. For chemical engineering process unit models there are one or two additional tips covered in Chapter 10.

3.3 PARAMETERIZING THE VESSEL MODEL

let's compute *metal_mass* vs. *H_to_D_ratio*

Reuse generally implies creating a model which will have as a part an instance of a previously defined type. For example, let us compute *metal_mass* as a function of the *H_to_D_ratio* for a vessel for a fixed *vessel_volume*. We would like to see if there is a value for the *H_to_D_ratio* for which the *metal_mass* is minimum for a vessel with a given *vessel_volume*. We might wonder if *metal_mass* goes to infinity as this ratio goes either to zero or infinity.

3.3.1 CREATING A PARAMETERIZED VERSION OF VESSEL

parameters indicate likely object sharing

To use instances of our model as parts in another model, we can parameterize it. We use parameterization to tell a future user that the parameters are objects he or she is likely to share among many different parts of a model. We wish to create a table containing different values of *H_to_D_ratio* vs. *metal_mass*. We can accomplish this by computing simultaneously several different vessels having the same *vessel_volume*, *wall_thickness* and *metal_density.* The objects we want to see and/or share for each instance of a vessel should include, therefore: *H_to_D_ratio*, *metal_mass*, *metal_density*, *vessel_volume* and *wall_thickness*.

The code in Figure 3-3 indicates the changes we make to the model declaration statement and the statements defining the variables to parameterize our model.

```
REQUIRE "atoms.a4l";
MODEL vessel(
   vessel_vol     "the volume contained within the cylindrical vessel"
      WILL_BE volume;
   wall_thickness "the thickness of all of the vessel walls"
      WILL_BE distance;
   metal_density  "density of the metal from which the vessel is constructed"
      WILL_BE mass_density;
   H_to_D_ratio   "the ratio of vessel height to diameter"
      WILL_BE factor;
   metal_mass     "the mass of the metal in the walls of the vessel"
      WILL_BE mass;
);
    NOTES
   'author' SELF        {Arthur W. Westerberg}
   'creation date' SELF {May, 1998}
    END NOTES;

   (* variables *)
   side_area      "the area of the cylindrical side wall of the vessel",
   end_area       "the area of the flat ends of the vessel"
                IS_A area;
   wall_vol       "the volume of the walls for the vessel"
                IS_A volume;
   H              "the vessel height (of the cylindrical side walls)",
   D              "the vessel diameter"
                IS_A distance;
   (* equations *)
FlatEnds:            end_area = 1{PI} * D^2 / 4;
```

```
Sides:             side_area = 1{PI} * D * H;
Cylinder:          vessel_vol = end_area * H;
Metal_volume:      (side_area + 2 * end_area) * wall_thickness = wall_vol;
HD_definition:     D * H_to_D_ratio = H;
VesselMass:        metal_mass = metal_density * wall_vol;

METHODS
METHOD specify;
    NOTES
   'purpose' SELF {to fix four variables and make the problem well-posed}
    END NOTES;
   vessel_vol.fixed        := TRUE;
   H_to_D_ratio.fixed      := TRUE;
   wall_thickness.fixed    := TRUE;
   metal_density.fixed     := TRUE;
END specify;
METHOD values;
    NOTES
   'purpose' SELF {to set the values for the fixed variables}
    END NOTES;
   H_to_D_ratio            := 2;
   vessel_vol              := 250 {ft^3};
   wall_thickness          := 5 {mm};
   metal_density           := 5000 {kg/m^3};
END values;
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 default_self;
   D                       := 1 {m};
   H                       := 1 {m};
END default_self;
METHOD default_all;
   RUN default_self;
   vessel_vol              := 1 {m^3};
   wall_thickness          := 5 {mm};
   metal_density           := 5000 {kg/m^3};
   H_to_D_ratio            := 1;
END default_all;
END vessel;
ADD NOTES IN vessel;
'description' SELF {This model relates the dimensions of a
           cylindrical vessel -- e.g., diameter, height and wall thickness
           to the volume of metal in the walls.  It uses a thin wall
           assumption -- i.e., that the volume of metal is the area of
           the vessel times the wall thickness.}
'purpose' SELF {to illustrate the insertion of notes into a model}
END NOTES;
```

Figure 3-3 The parameterized version of vessel model (vesselParams.a4c)

Substitute the statements in Figure 3-3 for lines 2 through 9 in Figure 3-2. Save the result in the file *vesselParam.a4c*.

Note the use of the WILL_BE statement in the parameter list. By declaring that the type of a parameter will be compatible with the types shown, the compiler can tell immediately if a user of this model is passing the wrong type of object when defining an instance of a vessel.

3.3.2 USING THE PARAMETERIZED VESSEL MODEL

Creating a table of metal_mass values vs. H_to_D_ratio

We next need to create a type definition that will set up our table of *H_to_D_ratio* values vs. *metal_mass* so we can observe approximately where it attains a minimum value. ASCEND allows us to create arrays of instances of any type. Here we shall create an array of vessels. The type definition is shown in Figure 3-4.

```
REQUIRE "atoms.a4l";
MODEL vessel(
   vessel_vol "the volume contained within the cylindrical vessel"
      WILL_BE volume;
   wall_thickness "the thickness of all of the vessel walls"
      WILL_BE distance;
   metal_density "density of the metal from which the vessel is constructed"
      WILL_BE mass_density;
   H_to_D_ratio "the ratio of vessel height to diameter"
      WILL_BE factor;
   metal_mass "the mass of the metal in the walls of the vessel"
      WILL_BE mass;
);
    NOTES
   'author' SELF {Arthur W. Westerberg}
   'creation date' SELF {May, 1998}
    END NOTES;
   (* variables *)
   side_area "the area of the cylindrical side wall of the vessel",
   end_area "the area of the flat ends of the vessel"
         IS_A area;
   wall_vol "the volume of the walls for the vessel"
         IS_A volume;
   H     "the vessel height (of the cylindrical side walls)",
   D     "the vessel diameter"
         IS_A distance;
   (* equations *)
FlatEnds:end_area = 1{PI} * D^2 / 4;
Sides:side_area = 1{PI} * D * H;
Cylinder:vessel_vol = end_area * H;
Metal_volume:(side_area + 2 * end_area) * wall_thickness = wall_vol;
HD_definition:D * H_to_D_ratio = H;
VesselMass:metal_mass = metal_density * wall_vol;
METHODS
METHOD specify;
    NOTES
   'purpose' SELF {to fix four variables and make the problem well-posed}
    END NOTES;
   vessel_vol.fixed     := TRUE;
   H_to_D_ratio.fixed   := TRUE;
   wall_thickness.fixed := TRUE;
   metal_density.fixed  := TRUE;
END specify;
METHOD values;
    NOTES
   'purpose' SELF {to set the values for the fixed variables}
    END NOTES;
   H_to_D_ratio         := 2;
   vessel_vol           := 250 {ft^3};
   wall_thickness       := 5 {mm};
   metal_density        := 5000 {kg/m^3};
END values;
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 default_self;
   D                    := 1 {m};
   H                    := 1 {m};
END default_self;
METHOD default_all;
   RUN default_self;
   vessel_vol           := 1 {m^3};
   wall_thickness       := 5 {mm};
   metal_density        := 5000 {kg/m^3};
   H_to_D_ratio         := 1;
END default_all;
END vessel;

ADD NOTES IN vessel;
'description' SELF {This model relates the dimensions of a
           cylindrical vessel -- e.g., diameter, height and wall thickness
           to the volume of metal in the walls.  It uses a thin wall
           assumption -- i.e., that the volume of metal is the area of
           the vessel times the wall thickness.}
'purpose' SELF {to illustrate the insertion of notes into a model}
END NOTES;

MODEL tabulated_vessel_values;
    vessel_volume    "volume of all the tabulated vessels"
         IS_A volume;
    wall_thickness   "thickness of all the walls for all the vessels"
         IS_A distance;
    metal_density    "density of metal used for all vessels"
         IS_A mass_density;
    n_entries        "number of vessels to simulate"
         IS_A integer_constant;
    n_entries :== 20;
    H_to_D_ratio[1..n_entries]   "set of H to D ratios for which we are
                     computing metal mass"
         IS_A factor;
    metal_mass[1..n_entries]     "mass of metal in walls of vessels"
         IS_A mass;
    FOR i IN [1..n_entries] CREATE
   v[i]              "the i-th vessel model"
         IS_A  vessel(vessel_volume, wall_thickness,
   metal_density, H_to_D_ratio[i], metal_mass[i]);
    END FOR;

METHODS
METHOD default_self;
END default_self;
```

```
METHOD specify;
   RUN v[1..n_entries].specify;
END specify;
METHOD values;
    NOTES 'purpose' SELF {to set up 20 vessel models having H to D ratios
        ranging from 0.1 to 2.}
    END NOTES;
   vessel_volume              := 250 {ft^3};
   wall_thickness             := 5 {mm};
   metal_density              := 5000 {kg/m^3};
   FOR i IN [1..n_entries] DO
      H_to_D_ratio[i]         := i/10.0;
   END FOR;
END values;
METHOD scale_self;
END scale_self;
END tabulated_vessel_values;

ADD NOTES IN tabulated_vessel_values;
'description' SELF {This model sets up an array of vessels to
           compute a range of metal_mass values for different values
           of H_to_D_ratio.}
'purpose' SELF {to illustrate the use of arrays in ASCEND}
END NOTES;
```

Figure 3-4 The code for the *tabulated_vessel_values* model (vesselTabulated.a4c)

Add this model to the end of the file *vesselParam.a4c* (after the vessel model) and save the file as *vesselTabulated.a4c*. Compile an instance of *tabulated_vessel_values* (call it *tvv*), run the *values* and *specify* methods for it, and then solve it. You will discover that the tenth element of the *metal_mass* array, corresponding to an *H_to_D_ratio* of *1* has the minimum value of *510.257 kilograms*.

3.4 CREATING A SCRIPT TO DEMONSTRATE THIS MODEL

The last step to make the model reusable is to create a script that anyone can easily run. Running the model successfully will allow a user to demonstrate the use of the model and to explore an instance it by browsing it.

ASCEND allows one to create such a script using either an editor or the tools in the **Script** window.

Restart the ASCEND system. You will have three windows open plus the large one which you can close by pressing its *dismiss* button. The **Script**, the **Library** and the **Console**[1] windows remain

In the **Script** window you will see the license agreement information for ASCEND. First get a new script buffer by selecting the *New file* tool under the *File* button.

Select the tool *Record actions* under the *Edit* button to start recording the steps you are about to undertake.

- In the **Library** window, under the *Edit* button, select *Delete all types*. Hit *Delete all* on the small confirmation window that appears.
- Load the file *vesselTabulated.a4c*, the file containing the model called *tabulated_vessel_values.* Do this by selecting the *Read types from file* tool under the *File* button and browsing the file system to find it. If you have trouble finding it, be sure to set the *Files of type* window at the bottom of the file browsing window to allow all types of files to be seen.
- Select the type *tabulated_vessel_values* in the right **Library** window and compile an instance of it by selecting the *Create simulation* tool under the *Edit* button. In the small window that appears, enter the name *tvv* and hit *OK*.
- Export the instance to the **Browser** by selecting the *Simulation to Browser* tool under the *Export* button.
- Initialize the variable values by running the *values* method. Do this by selecting the *Run method* tool under the *Edit* button. Select the *values* method and hit *OK*.
- Set the fixed flags to get a well-posed problem by repeating the last step but this time select the *reset* method.
- Export the instance *tvv* to the **Solver** by selecting the *to Solver* tool under the *Export* button.
- Solve tvv by selecting the *Solve* tool under the *Execute* button in the **Solver** window.
- Return to the **Script** window and turn off the recording by selecting the *Record actions* tool under the *Edit* button.

1. UNIX users should treat the xterm where they started ASCEND as their **Console** window.

- Save the script you have just created by selecting the *Save* tool under the *File* button of the **Script** window. Name the file *vesselTabulated.a4s* (note the '*s*' ending) to indicate it is the script to run an example problem for models in the *vesselTabulated.a4c* (note the '*c*' ending) file.
- Exit ASCEND by selecting the *Exit ASCEND* tool under the *File* button on the **Script** window. The contents of the **Script** window will be similar to that in Figure 3-5 (the path to the file may differ).
- Restart ASCEND.
- Open the script you just created by selecting the *Read file* tool under the *File* button on the **Script** window. (Be sure you are allowing the system to see files with the ending *a4s* by setting the *Files of type* window at the bottom of the file browsing window.)
- Highlight all the instructions in this script and then execute the highlighted instructions by selecting the *Statements selected* tool under the *Execute* button.

You will run the same sequence of instructions you ran to create the script.

```
DELETE TYPES;
READ FILE "vesselTabulated.a4c";
COMPILE tvv OF tabulated_vessel_values;
BROWSE {tvv};
RUN {tvv.reset};
RUN {tvv.values};
SOLVE {tvv} WITH QRSlv;
```

Figure 3-5 Script to run vesselTabulated.a4c (this is the contents of the file vesselTabulated.a4s)

3.4.1 DISCUSSION

In this chapter we converted the vessel model into a form where you and others in the future will have a chance to reuse it. We did this by first adding methods to make the problem well-posed and to provide values for the fixed variables for which we readily found a solution when playing with our original model as we did in the previous chapter. We then thought of a typical use for this model and developed a parameterized version based on that use. If this model were in a library, a future user of it would most often simply have to understand the parameters to create an instance of this type of model. We next added

NOTES, a form of active comments, to the model. We suggest that notes are much more useful than comments as we can provide tools that can extract them and allow us to search them, for example, to find a model with a given functionality. Finally, we showed you how to create a script by turning on a “phone” session where ASCEND records the actions one takes when loading, compiling and solving a model. One can save and play this script in the future to see a typical use of the model.

In the next chapter, we look at how we can plot the results we created in the model vesselTabulated.a4c. We will have to reuse a model someone else has put into the library of available models. In other words, the “shoe is on the other foot,” and we quickly experience the difficulties with reuse first hand. We will also learn how to run a case study from which we can extract the same information with a single vessel model run multiple times.

CHAPTER 4 CREATING A PLOT (USING A LIBRARY MODEL)

In this chapter we are going to produce a plot by using a model that someone else has created. We gain two lessons: (1) you will understand first hand the difficulties one encounters when trying to use a model someone else has created and (2) you will learn how to produce a plot in ASCEND. The approach we take is not the one you should take if your goal is simply to produce this plot. Our goal is pedagogical, not efficiency. In the last chapter we created an array of vessel models to produce the data that we now about to plot. We approached this problem this way so you could see how one creates arrays in ASCEND. Having this model, we have the data. The easiest thing we can do now it use it to produce a plot.

We also have in ASCEND the ability to do case studies over a model instance, varying one or more of the fixed variables for it over a range of values and capturing the values of other variables that result. This powerful case study tool is the proper way to produce this plot as ASCEND only has to compile one instance and solve it repeatedly rather than produce an array of models. We finish this chapter showing you how to use this case study tool.

4.1 CREATING A PLOT

We want a plot of *metal_mass* values vs. *H_to_D_ratio*. If we look around at the available tools, we find there is a *Plot* tool under the *Display* button in the **Browser** window. While not obvious, it turns out we can plot the arrays we produce when we include instances of type *plt_plot_integer* and *plt_plot_symbol* in our model. We find these types in the file *plot.a4l* located in the ASCEND4 models directory which is distributed with ASCEND. Figure 4-1 is a print out of that file (but with line numbers added so we can reference them here).

```
REQUIRE "system.a4l";                                                      1
PROVIDE "plot.a4l";                                                        2
(*******************************************************************\      3
   plot.a4l                                                                4
   by Ben Allan                                                            5
   Part of the Ascend Library
This file is part of the Ascend modeling library.
Copyright (C) 1997 Benjamin Andrew Allan
The Ascend modeling library is free software; you can redistribute
it and/or modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.
The Ascend Language Interpreter is distributed in hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
You should have received a copy of the GNU General Public License along with
the program; if not, write to the Free Software Foundation, Inc., 675
Mass Ave, Cambridge, MA 02139 USA.  Check the file named COPYING.
\*********************************************************************)
(*********************************************************************\
  $Date: 97/08/04 15:22:21 $
  $Revision: 1.1 $
  $Author: ballan $
  $Source: /afs/cs.cmu.edu/project/ascend/Repository/models/plot.a4l,v $
\*********************************************************************)
(*===========================================================================
==*
    P L O T . A 4 L
    ---------------
   AUTHOR:Ben Allan
      provoked by plot.lib by Peter Piela and Kirk A. Abbott
   DATES:03/97 - Original code.
   CONTENTS:
      A parameterized plot library mostly compatible
      with plot.lib, but with variable graph titles.
*)
MODEL pltmodel() REFINES cmumodel();
END pltmodel;
MODEL plt_point(
   x WILL_BE real;
   y WILL_BE real;
) REFINES pltmodel();
END plt_point;
(***************************************************************)
MODEL plt_curve(
   npnts IS_A set OF integer_constant;
   y_data[npnts] WILL_BE real;
   x_data[npnts] WILL_BE real;
) REFINES pltmodel();
(* points of matching subscript will be plotted in order of
 * increasing subscript value.                                  51
 *)                                                             52
   legend      IS_A symbol; (* mutable now! *)                  53
   FOR i IN [npnts] CREATE                                      54
      pnt[i]IS_A plt_point(x_data[i],y_data[i]);                55
   END FOR;                                                     56
END plt_curve;                                                  57
(*************************************************************) 58
MODEL plt_plot_integer(                                         59
   curve_set IS_A set OF integer_constant;                      60
   curve[curve_set] WILL_BE plt_curve;                          61
) REFINES pltmodel();                                           62
   title, XLabel, YLabel IS_A symbol; (* mutable now! *)        63
   Xlow IS_A real;                                              64
   Ylow IS_A real;                                              65
   Xhigh IS_A real;                                             66
   Yhigh IS_A real;                                             67
   Xlog IS_A boolean;                                           68
   Ylog IS_A boolean;                                           69
END plt_plot_integer;                                           70
(*************************************************************) 71
MODEL plt_plot_symbol(                                          72
   curve_set IS_A set OF symbol_constant;                       73
   curve[curve_set] WILL_BE plt_curve;                          74
) REFINES pltmodel();                                           75
   title, XLabel, YLabel IS_A symbol; (* mutable now! *)        76
   Xlow IS_A real;                                              77
   Ylow IS_A real;                                              78
   Xhigh IS_A real;                                             79
   Yhigh IS_A real;                                             80
   Xlog IS_A boolean;                                           81
   Ylog IS_A boolean;                                           82
END plt_plot_symbol;                                            83
```

Figure 4-1 The file plot.a4l

As you can see, this file contains the two types we seek—starting in lines 59 and 72, respectively. However, before we can use them, we do need to understand them. We are, so to speak, on the receiving end of the reusability issue. If you spend some time, you will find that you can decipher these model definitions. To make that less painful, we will help you do so here. If these models were better documented, they would be much less difficult to interpret. In time we will add Notes to them to remedy this deficiency.

4.1.1 MODEL REFINEMENT

please, explain “refines”

The first model, pltmodel, is two lines long, having a *MODEL* statement indicating it “refines” *cmumodel* and an *END* statement. We have not encountered the concept of refinement as yet. In ASCEND the “refines” means the model *pltmodel* inherits all the statements of *cmumodel*, a model which has been defined at the end of the file *system.a4l*. We show the code for *cmumodel* in Figure 4-2, and we note that it too is an empty model. It is, as it says, a root for a collection of loosely related models. You will note (and forgive) a bit of dry humor by its author, Ben Allan. So far as we know, this model neither provokes nor hides any bugs.

```
MODEL cmumodel();
(* This MODEL does nothing except provide a root
 * for a collection of loosely related models.
 * If it happens to reveal a few bugs in the software,
 * and perhaps masks others, well, what me worry?
 * BAA, 8/97.
 *)
END cmumodel;
```

Figure 4-2 The code for cmumodel

We need to introduce the concept of type refinement to understand these models. We divert for a moment to do just that.

parents and children in a refinement hierarchy

Suppose model *B* refines model *A*. We call *A* the parent model and *B* the child. The child model *B* inherits all the code defining the parent model *A*. In writing the code for model *B*, we do not write the code it inherits from *A*; we simply understand it is there already. The code we write for model *B* will be only those statements that we wish to add beyond the code defining its parent. ASCEND supports only single inheritance; thus a child may have only one parent. A parent, on the other hand, may have many children, each inheriting its code.

order does not matter in declarative code

We are dealing in ASCEND with models defined by their variables and equations. As we have noted above, the order for the statements defining each of these does not matter—i.e., the variables and equations may be defined in any order. So adding new variables and equations through refinement may be done quite easily.

but it does in the procedural code for methods

In contrast, the methods are bits of procedural code—i.e., they are run as a sequence of statements where order does matter. In ASCEND, a child model will inherit all the methods of the parent. If you wish to alter the code for a method, you must replace it entirely, giving it the same name as the method to be replaced. (However, if you look into the documentation on the methods (syntax.pdf), you will find that the original method is still available for execution. You simply have to add a qualifier to its name to point to it.)

If we look into this file we see the refinement hierarchy shown in Figure 4-3. *cmumodel* is the parent model for all these models. *pltmodel* is its child. The remaining three models are children of *pltmodel*.

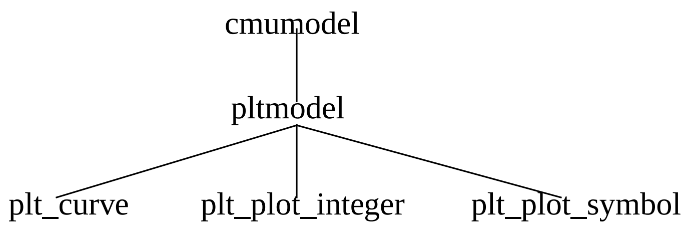

Figure 4-3 The refinement hierarchy in the file plot.a4l

(We can have ASCEND show us the refinement hierarchy. From the **Library** window, select *Read types from file* from the *File* button, and click on *plot.a4l* (you may need to change the filter to see the *.a4l* files). Select *plot.a4l* from the left hand-pane of the **Library**, and then *plt_plot_symbol* from the right-hand pane. Finally, choose the *Ancestry* tool from the *Display* button.)

There are three reasons to support model refinement, with the last being the most important one.

reasons for refinement

- **We write more compact code**: The first reason is compactness of coding. One can inherit a lot of code from a parent. Only the new statements belonging to the child are then written to define it. This is not a very important reason for having refinement.
- **Changes we make to the parent propagate**: A second reason is that one can edit changes into the parent and know that the children will inherit those changes without having to alter the code written for the child. (Of course, one can change the parent in such a way that the changes to the child are not what is wanted for the child, introducing what will likely become some interesting debugging problems.)

with the most important being we know what can substitute for what

- **We know what can substitute for what**: The most important reason is that inheritance tells us what kinds of parts may be substituted for a particular part in a model. Because a child inherits all the code from its parent, we know the child has all the variables and equations defined for it that the parent does—and typically more. **We can use an instance of the child as a replacement for an instance of the parent.** Thus if you were to write a model with the part *A1* of type *A* in it, someone else can create an instance of your model and substitute a part *B1* which is of type *B*. This substituted part will have all the needed variables in it that you assumed would be there.

This third reason says that when a object passed as a parameter WILL_BE of type *A*, we know that a part of either type *A* or type *B* will work.

4.1.2 CONTINUING WITH CREATING A PLOT

We are going to include in our model a part of type *plt_plot_integer* or *plt_plot_symbol* that ASCEND can plot. We need to look at the types of parameters required by whichever of these two we select to include here. Tracing back to its parents, we see them to be empty so all the code for these types is right here.

The first parameter we need is a *curve_set* which is defined to be a set of *integer_constant* or of *symbol_constant*. We have to guess at this time at the purpose for *curve_set*. It would really help to have notes defining the intention here and to have a piece of code that would demonstrate the use of these models. At present, we do not. We proceed, admitting we will appear to “know” more than we should about this model. It turns out that *curve_set* allows us to identify each of the curves we are going to plot. These models assume we are plotting several variables (let’s call them *y[1]*, *y[2]*, ...) against the same independent variable *x*. The values for curve_set are the ‘1’, ‘2’, etc. identifying these curves.

Here we wish to plot only one curve presenting *metal_mass* vs. *H_to_D_ratio*. We can elect to use *plt_plot_symbol* and label this curve ‘*5 mm*’. The label ‘*5 mm*’ is a *symbol* so we will create a set of type *symbol* with this single member.

The second object has to be a object of type *plt_curve*.

Looking at line 45, we see how to include an object of type *plt_curve*. It must be passed three objects: a set of integers (e.g., the set of integers from *1* to *20*) and two lists of data giving the *y*-values vs. the *x*-values for the curve. In the model *tabulated_vessel_values*, we have just these two lists, and they are named *metal_mass* and *H_to_D_ratio*.

In Figure 4-4, we show the code you need to add to the model *tabulated_vessel_values*. It contains a part called *massVSratio* of type *plt_plot_symbol* that ASCEND can plot. This code is at the end of the declarative statements in tabulated_vessel_values. It also replaces the first method, METHOD default_self.

```
    CurveSet "the index set for all the curves to be plotted"
            IS_A set OF symbol_constant;
    CurveSet :== ['5 mm'];

   Curves['5 mm'] "the one curve of 20 points for metal_mass vs. H_to_D_ratio"
            IS_A plt_curve([1..n_entries], metal_mass, H_to_D_ratio);
    massVSratio "the object ASCEND can plot"
            IS_A plt_plot_symbol(CurveSet, Curves);

METHODS
METHOD default_self;
(* set the title for the plot and the labels for the ordinate and abscissa *)
    massVSratio.title :=
     'Metal mass of the walls vs H to D ratio for a thin-walled cylindrical
vessel';
    massVSratio.XLabel := 'H to D ratio';
    massVSratio.YLabel := 'metal mass IN kg/m^3';
END default_self;
```

Figure 4-4 The last bit of new code to include a plot in the model *tabulated_vessel_values* (save as vesselPlot.a4c)

Also just after the first line in this file—which reads

```
REQUIRE “atoms.a4l”;
```

place the instruction

```
REQUIRE “plot.a4l”;
```

When you solve this new instance and make *massSVratio* the current object, you will find the *Plot* tool under the *Display* button in the **Browser** window lights up and can be selected. If you do this, you will get a plot of *metal_mass* vs. *H_to_D_ratio*. A clear minimum is apparent on this plot at *H_to_D_ratio* equal to approximately one.

You should create a script to run this model just as you did for *vesselTabulated.a4c* in the previous chapter. Save it as *vesselPlot.a4s*.

4.2 CREATING A CASE STUDY FROM A SINGLE VESSEL

You may think creating an array of vessels and a complex plot object just to generate a graph is either an awful lot of work or a method which will not work for very large models. You think correctly on both points. The plt_plot models are primarily useful for sampling values from an array of inter-related models that represent a spatially distributed system such as the pillars in a bridge or the trays in a distillation column. You can conduct a case study, solving a single model over a range of values for some specified variable, using the Script command STUDY.

We will step through creating a base case and a case study using the vessel model. Start by opening a new buffer in the Script window and turning on the record button of the Script's edit menu. In the Library window run the "Delete all types" button to clear out any previous simulations. Load the vessel model from the file vesselMethods.a4c you created in Section 3.2.

4.2.1 THE BASE CASE

compile a vessel.

Select and compile the vessel model. Give the simulation the name `V`. Select the simulation `V` in the bottom pane of the Library window and use the right mouse button (or Alt-x b) to send the simulation to the Browser.

solving the base case.

In the Browser, place the mouse cursor over the upper left pane. Use the right mouse button to run methods `reset` and `values`, then send the model to the Solver by typing "Alt-x s". Move the mouse to the Solver window and hit the F5 key to solve the model.

graphical case study optimization

We now know that it takes 535.7 kg of metal to make a 250 cubic foot vessel which is twice as high as it is broad. Suppose that now we want to know the largest volume that this amount of metal can contain assuming the same wall thickness is required. Perhaps a skinnier or fatter vessel can hold more, so we need to do a case study using the aspect ratio (`H_to_D_ratio`) as the independent variable. Use the Browser to change `V.metal_mass.fixed` to TRUE, since we are using a constant amount of metal. The solver will want you to free a variable now, so select `V.vessel_vol` to be freed, since volume is what we want to study.

script recorded so far

Turn off the recording button on the Script window. The recording should look something like

```
DELETE TYPES;
READ FILE {vesselMethods.a4c};
COMPILE V OF vessel;
BROWSE {V};
RUN {V.reset};
SOLVE {V} WITH QRSlv;
ASSIGN {V.metal_mass.fixed} TRUE {};

# you must type the next line in the script yourself.
ASSIGN {V.vessel_vol.fixed} FALSE {};
```

The file ascend4/models/vesselStudy.a4s was recorded in a similar manner.

4.2.2 CASE STUDY EXAMPLES

configuring a case study

The STUDY command takes a lot of arguments. We'll explain them all momentarily, but should you forget them simply enter the command `STUDY` without arguments in the ASCEND Console window or xterm window to see an error message explaining the arguments and giving an example. Enter the following command in the Script window exactly as shown except for the file name following `OUTFILE`. Specify a file to be created in *your* ascdata directory.

```
STUDY {vessel_vol} \
IN {V} \
VARYING {{H_to_D_ratio} {0.1} {0.5} {0.8} {1} {1.5} {2} \
{3} {4} {8}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
```

This is the simplest form of case study; the backslashes at the end of each line mean that it is all one big statement. Select all these lines in the Script at once with the mouse and then hit F5 to execute the study. The solver will solve all the cases and produce the output file vvstudy.dat. The quickest way to see the result is to enter the following command in the Script, then select and execute it. (Remember to use the name of your file and not the name shown).

```
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
ASCPLOT CLOSE; #omit if you want to see data table
```

You should get a graph that looks something like Figure 4-5. The largest volume is in the neighborhood of an `H_to_D_ratio` of 1.

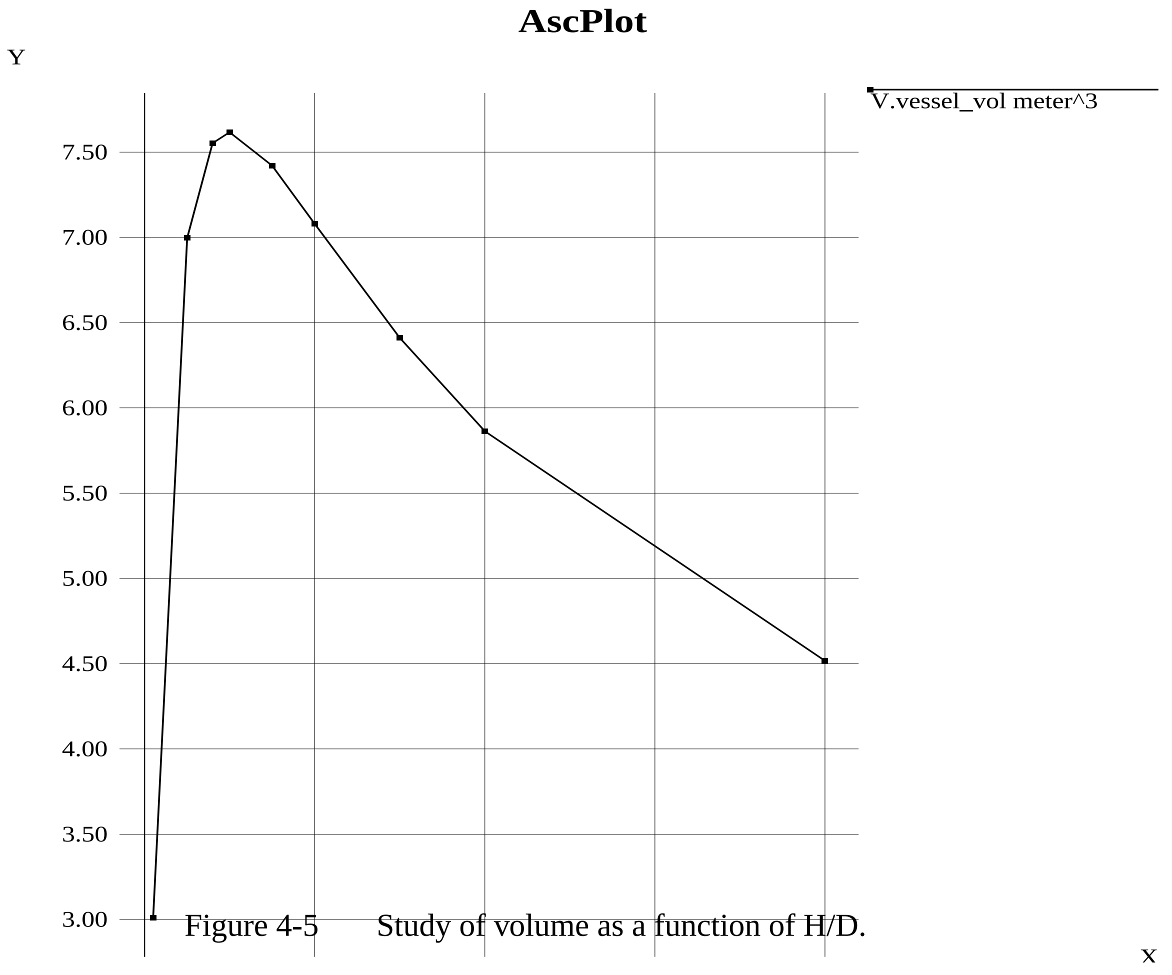

Figure 4-5 Study of volume as a function of H/D.

4.2.2.1 MULTI-VARIABLE STUDIES

We now have an idea where the solution is most interesting, so we can do a detailed study where we also monitor other variables such as surface areas. Additional variables to watch can be added to the STUDY clause of the statement.

```
STUDY {vessel_vol} {end_area} {side_area} \
IN {V} \
VARYING {{H_to_D_ratio} {0.5} {0.6} {0.7} {0.8) {0.9} \
{1} {1.1} {1.2} {1.3}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
ASCPLOT CLOSE; #omit if you want to see data table
```

4.2.2.2 MULTI-PARAMETER STUDIES

We can also do a multi-parameter study, for example also varying the wall thickness allowed. In general, any number of the fixed variables can be varied in a single study, but be aware that ASCEND's relatively simple plotting capabilities do not yet include surface or contour maps so you will need another graphic tool to view really pretty pictures.

```
STUDY {vessel_vol} \
IN {V} \
VARYING \
{{H_to_D_ratio} {0.8) {0.9} {1} {1.1} {1.2} {1.3}} \
{{wall_thickness} {4 {mm}} {5 {mm}} {6 {mm}} {7 {mm}}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
```

In this study the peak volume occurs at the same `H_to_D_ratio` for any wall thickness but the vessel volume increases for thinner walls. This may be hard to see with the default graph settings, but column 2 in rows 8-11 (H_to_D = 1.0) of the ASCPLOT data table have the largest volumes for any given thickness in column 1. Notice that the units must be specified for the `wall_thickness` values in the VARYING clause.

4.2.2.3 PLOTTING OUTPUT WITH OTHER TOOLS

To convert the study results from the ASCPLOT format to a file more suitable for importing into a spreadsheet, the following command does the trick. As usual, change the names to match your `ascdata` directory.

```
asc_merge_data_files excel \
{/usr0/ballan/ascdata/vvs.txt} \
{/usr0/ballan/ascdata/vvstudy.dat}
```

If you prefer Matlab style text, substitute 'matlab' for 'excel' in the line above and change the output name from 'vvs.txt' to 'vvs.m'.

4.2.3 STUDY BEHAVIOR DETAILS

variable list

We now turn to the details of the STUDY statement. As we saw in Section 4.2.2.1, any number of variables to be monitored can follow the STUDY keyword.

IN clause

The IN clause specifies which part of a simulation is to be sent to the Solver; a small part of a much larger model can be studied if you so desire. All the variable and parameter names that follow the STUDY keyword and that appear in the VARYING clause must be found in this part of the simulation.

parameter list

The VARYING clauses is a list of lists. Each inner list gives the name of the parameter to vary followed by its list of values. Each possible combination of parameter values will be attempted in multi-parameter studies. If a case fails to solve, then the study will behave according to the option set in the ERROR clause.

solver name

The solver named in the USING clause is invoked on each case. The solver may be any of the algebraic solvers or optimizers, but the integrators (e.g. LSODE) are not allowed.

data file name

The case data are stored in the file name which appears in the OUTFILE clause. By default, this file is overwritten when a STUDY is started, so if you want multiple result files, use separate file names.

error handling

When the solver fails to converge or encounters an error, the STUDY can either ignore it (`ERROR IGNORE`) and go on to the next case, warn you (`ERROR WARN`) and go on to the next case, or stop (`ERROR STOP`). The ERROR option makes it possible start a case study and go to lunch. Cases which fail to solve will not appear in the output data file.

Note that if the model is numerically ill-behaved it is possible for a case to fail when there is in fact a solution for that combination of parameters. STUDY uses the solution of the last successfully solved case as the initial guess for the next case, but sometimes this is not the best strategy. STUDY also does not attempt to rescale the problem from case to case. When a case that you think should succeed fails, go back and investigate that region of the model again manually or with a more narrowly defined study.

4.3 DISCUSSION

We have just led you step by step through the process of creating, debugging and solving a small ASCEND model. We then showed you how to make this model more reusable, first by adding comments and methods. Methods capture the “how you got it well-posed” experience you had when first solving an instance of the vessel model. We then showed you how to parameterize this model and then use it to construct a table of *metal_mass* values vs. *H_to_D_ratio* values. Finally we showed you how to add a plot of these results. You should next look at the chapter in the documentation where you create two more small ASCEND models. This chapter gives you much less detail on the buttons to push. Finally, if you are a chemical engineer, you should look at the chapter on the script and model for a simple flowsheet (simple_fs.a4s and simple_fs.a4c respectively).

With this experience you should be ready to write your own simple ASCEND models to solve problems that you might now think of solving using a spreadsheet. Remember that once you have the model debugged in ASCEND, you can solve inside out, backwards and upside down and NOT just the way you first posed it—unlike your typical use of a spreadsheet model.

CHAPTER 5 MANAGING MODEL DEFINITIONS, LIBRARIES, AND PROJECTS

Most complex models are built from parts in one or more libraries. In this chapter we show typical examples of how to make sure your model gets the libraries it needs. We then explain in more general terms the ASCEND mechanism which makes this work and how you can use it to manage multiple modeling projects simultaneously.

5.1 USING REQUIRE AND PROVIDE

5.1.1 REQUIREING SYSTEM.A4L

Suppose you are in a great hurry and want to create a simple model and solve it without concern for good style, dimensional consistency, or any of the other hobgoblins we preach about elsewhere. You will write equations using only *generic_real* variables as defined in system.a4l. The equations in this example do not necessarily have a solution. In your ascdata (see howto1) directory you create an application model definition file “myfile.a4c” which looks like:

```
REQUIRE “system.a4l”;
MODEL quick_n_dirty;
   x = y^2;
   y = x + 2*z;
   z = cos(x+y);
   x,y,z IS_A generic_real;
(* homework problem 3, due May 21. *)
END quick_n_dirty;
```

The very first line ‘REQUIRE “system.a4l”; tells ASCEND to find and load a file named “system.a4l” if it has not already been loaded or provided in some other way. This REQUIRE statement must come before the MODEL which uses the *generic_real* ATOM that system.a4l defines.

The REQUIRE statements in a file should all come at the beginning of the file before any other text, including comments. This makes it very easy for other users or automated tools to determine which files, if any, your models require.

On the ASCEND command line (in the Console window or xterm) or in the Script window, you can then enter and execute the statement

```
   READ FILE “myfile.a4c”;
```

to cause system.a4l and then myfile.a4c to be loaded.

5.1.2 CHAINING REQUIRED FILES

Notice when you read myfile.a4c that ASCEND prints messages about the files being loaded. You will see that a file “basemodel.a4l” is also loaded. In system.a4l you will find at the beginning the statements

```
   REQUIRE “basemodel.a4l”;
   PROVIDE “system.a4l”;
```

The basemodel library is loaded in turn because of the REQUIRE statement in system.a4l. We will come back to what the PROVIDE statement does in a moment. This chaining can be many files deep. To see a more complicated example, enter

```
   READ FILE column.a4l;
```

and watch the long list of files that gets loaded. If you examine the first few lines of each file in the output list, you will see that each file REQUIRES only the next lower level of libraries. This style minimizes redundant loading messages and makes it easy to substitute equivalent libraries in the nested lower levels without editing too many higher level libraries. The term “equivalent libraries” is defined better in the later section on PROVIDE.

5.1.3 BETTER APPLICATION MODELING PRACTICE

never require system.a4l in an application model.

It is generally a bad idea to create a model using only generic_real variables. The normal practice is to use correct units in equations and to use dimensional variables. In the following file we see that this is done by requiring “atoms.a4l” instead of “system.a4l” and by using correct units on the coefficients in the equations.

```
REQUIRE “atoms.a4l”; MODEL quick_n_clean;
   x = y^2/1{PI*radian};
   y = x + 2{PI*radian}*z;
   z = cos(x+y);
   x, y IS_A angle;
   z IS_A dimensionless;
(* homework problem 3, due May 21. *)
END quick_n_clean;
```

5.1.4 SUBSTITUTE LIBRARIES AND PROVIDE

ASCEND keeps a list of the already loaded files, as we hinted at in Section 5.1.1. A library file should contain a PROVIDE statement, as system.a4l does, telling what library it supplies. Normally the PROVIDE statement just repeats the file name, but this is not always the case. For example, see the first few lines of the file ivpsystem.a4l which include the statement

```
   PROVIDE “system.a4l”;
```

indicating that ivpsystem.a4l is intended to be equivalent to file system.a4l while also supplying new features. When ivpsystem.a4l is loaded both “system.a4l” and “ivpsystem.a4l” get added to the list of already loaded files. For one explanation of when this behavior might be desirable, see Section 12.1. Another use for this behavior is when creating and testing a second library to eventually replace the first one.

When a second library provides compatible but extended definitions similar to a first library, the second can be substituted for the first one. The second library will obviously have a different file name, but there is no need to load the first library if we already have the second one loaded. ivpsystem.a4l is a second library substitutable for the first library system.a4l. Note that the reverse is not true: system.a4l does not

```
   PROVIDE “ivpsystem.a4l”;
```

so system is not a valid substitute for ivpsystem.

5.1.5 REQUIRE AND COMBINING MODELING PACKAGES

Model libraries frequently come in interrelated groups. For example, the models referred to in Ben Allan’s thesis are published electronically as a package models/ben/ in ASCEND IV release 0.9. To use Ben’s distillation libraries, which require rather less memory than the current set of more flexible models, your application model should have the statement

```
   REQUIRE “ben/bencolumn.a4l”;
```

at the beginning.

Combining models from different packages may be tricky if the package authors have not documented them well. Since all packages are open source code which you can copy into your ascdata directory and modify to suit your needs, the process of combining libraries usually amounts to changing the names of the conflicting model definitions in your copy.

Do NOT use \ instead of / in the package name given to a REQUIRE statement even if you are forced to use Microsoft Windows.

5.2 HOW REQUIRE FINDS THE FILES IT LOADS

The file loading mechanism of REQUIRE makes it simple to manage several independent sets of models in simultaneous development. We must explain this mechanism or the model management may seem somewhat confusing. When a REQUIRE statement is processed, ASCEND checks in a number of locations for a file with that name: ascdata, the current directory, and the ascend4/models directory. We will describe how you can extend this list later. ASCEND also looks for model packages in each of these same locations.

5.2.1 ASCDATA

If your ascdata directory exists and is readable, ASCEND looks there first for required files. Thus you can copy one of our standard libraries from the directory ascend4/models to your ascdata directory and modify it as you like. Your modification will be loaded instead of our standard library. See Section 2.2 for how to find your ascdata directory.

5.2.2 THE CURRENT DIRECTORY

The current directory is what you get if you type ‘pwd’ at the ASCEND Console or xterm prompt. Under Microsoft Windows, the current directory is usually some useless location. Under UNIX, the current directory is usually the directory from which you started ASCEND.

5.2.3 ASCEND4/MODELS/

The standard (CMU) models and packages distributed with ASCEND are found in the ascend4/models/ subdirectory where ASCEND is installed. This directory sits next to the directory ascend4/bin/ where the ascend4 or ascend4.exe executable is located.

5.2.4 MULTIPLE MODELING PROJECTS

If you dislike navigating multi-level directories while working on a single modeling project, you can separate projects by keeping all files related to your current project in one directory and changing to that directory before starting ASCEND. If you have files that are required in all your projects, keep those files in your ascdata directory. Under Windows, cd to the directory containing the current project from the Console window after starting ASCEND.

5.2.5 EXAMPLE: FINDING "BEN/BENCOLUMN.A4L"

Suppose an application model requires bencolumn.a4l from package ben as shown in Section 5.1.5. Normally ASCEND will execute this statement by searching for:

```
~/ascdata/ben/bencolumn.a4l
./ben/bencolumn.a4l
$ASCENDDIST/ascend4/models/ben/bencolumn.a4l
```

Assuming we started ASCEND from directory /usr1/ballan/projects/test1 under UNIX, the full names of these might be

```
/usr0/ballan/ascdata/ben/bencolumn.a4l
/usr1/ballan/projects/test1/ben/bencolumn.a4l
/usr/local/lib/ascend4/models/ben/bencolumn.a4l
```

Assuming we started ASCEND from some shortcut on a Windows desktop, the full names of these locations might be

```
C:\winnt\profiles\ballan\ascdata\ben\bencolumn.a4l
C:\Program Files\netscape\ben\bencolumn.a4l
C:\ASCEND\ascend4\models\ben\bencolumn.a4l
```

The first of these three which actually exists on your disk will be the file that is loaded.

5.2.6 HOW REQUIRE HANDLES FILE AND DEFINITION CONFLICTS

Normally you simply delete all types before loading a new or revised set of ASCEND models and thus you avoid most conflicts. When you are working with a large simulation and several smaller ones, you may not want to delete all the types, however. We decided to make

REQUIRE handle this situation and the almost inevitable redundant REQUIRE statements that occur in the following reasonable way.

When a file is REQUIREd, ASCEND first checks the list of loaded and provided files for a name that matches. If the name is found, then that file is checked to see if it has changed since it was loaded. If the file has changed, then any definition that was changed is loaded in the ASCEND Library and the new definition is used in building any subsequently compiled simulations. Old simulations remain undisturbed and are not updated to use the new definitions since there may be conflicts that cannot be automatically resolved.

5.2.7 EXTENDING THE LIST OF SEARCHED DIRECTORIES

ASCEND uses the environment variable ASCENDLIBRARY as the list of directory paths to search for required files. Normally you do not set this environment variable, and ASCEND works as described above.

To see or change the value of ASCENDLIBRARY that ASCEND is using, examine ASCENDLIBRARY in the System utilities window available from the Script Tools menu. Changes made to environment variables in the utilities window are NOT saved. If you are clever enough to set environment variables before running ASCEND, you can make it look anywhere you want to put your model files. Consult your operating system guru for information on setting environment variables if you do not already know how.

CHAPTER 6 PLOTTING DATA SAMPLED FROM COMPLEX MODELS

Often you need a plot of data sampled from arbitrary locations in a model that are not naturally grouped in a single easily plotted vector. The plot.a4l library provides models (plt_curve, plt_plot_symbol, and plt_plot_integer) that can be used with the Browser's Display Plot button. In this chapter we see how to create such a plot using the ASCEND statement ALIASES/IS_A to sample data from a mechanical system of stretched springs, masses, anchors, and fingers. Creating plots of time series data output from ASCEND's initial value solver LSODE is discussed in Section 12.3, ”Viewing Simulation Results,” on page 125.

Chemical engineers who can tolerate distillation models should visit the file plotcol.a4c in the models library for more complicated examples of plotting and visit the model *simple_column_profiles* in column.a4l for more complicated examples of sampling data. Reading this chapter first may be of help in interpreting those models.

6.1 THE GRAPH WE WANT

We want to plot the positions X1 to X3 of the connecting hooks h1, h2, and h3 in a mechanical system as shown in Figure 6-1. The anchor,

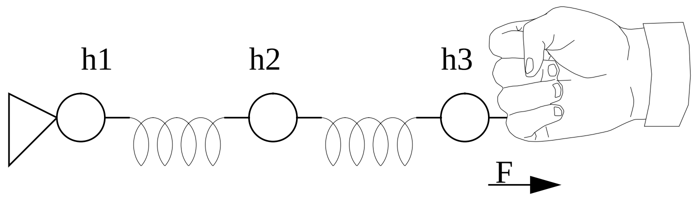

Figure 6-1 Spring test model system, st.

hooks, springs, and finger (we could replace either spring with a block mass, also) are all separate objects which we have modeled very simply. These models are given at the end of the chapter and can also be found (with improvements) in force1d.a4c, a model file in the distributed ASCEND libraries.

Plotting is usually a post-solution analysis tool, so our plots should not be entangled with the basic models or with the total mechanical system model, *st*. We might want to explain the system *st* to someone and this could be hard to do if the code is cluttered up with plot information.

6.2 CONSTRUCTING A PLOT CURVE

The plot library models follow object-oriented thinking carefully, perhaps a little too carefully. A plt_plot_integer is a plottable model built out of plt_curves which are in turn built out of arrays of data points from the user. Constructing these data arrays is the only significant challenge in using the plot models. Begin by building a new model with the system *st* as a part:

```
MODEL plot_spring_test;
   st IS_A spring_test;
   Plot_X IS_A plt_plot_integer(curve_set,curves);
END plot_spring_test;
```

We want to create a plt_curve from the array of hook numbers y_data[1..3] plotted against horizontal hook position x_data[1..3]. There are obvious problems with the model above: *curves* and *curve_set* are used without being defined and there is no mention of x_data or y_data.

Begin by using an ALIASES/IS_A statement to construct the array of positions x_data from the variables X stored in the hooks of model *st*.

```
x_data[Xset] ALIASES (st.h1.X,st.h2.X,st.h3.X) WHERE Xset
IS_A set OF integer_constant;
```

This statement creates a set, Xset, indexing a new array x_data with elements collected from st. Since the value of Xset is not specified, it becomes by default the set [1,2,3].

Now we need the hook numbers, y_data. These do not exist in st, so we create them. We will set the numeric values of these in the *default_self* method. We will include method in the final model, but do not show it here.

```
y_data[Xset] IS_A real;
```

Having both y_data and x_data, we can construct a curve from them:

```
X_curve IS_A plt_curve(Xset,y_data,x_data);
```

6.3 CONSTRUCTING THE ARRAY OF CURVES

We have a curve, but the plt_plot_integer model Plot_x expects an array of curves and the set indexing this array as input. We can make both from X_curve easily using, once again, an ALIASES/IS_A statement.

```
curves[curve_set] ALIASES (X_curve) WHERE curve_set IS_A
set OF integer_constant;
```

All the pieces are now in place, so we have the final model:

```
MODEL plot_spring_test;

   (* create our system model and plot. *)
   st IS_A spring_test;
   Plot_X IS_A plt_plot_integer(curve_set,curves);

   (* Gather the sampled data into an array *)
   x_data[Xset] ALIASES (st.h1.X,st.h2.X,st.h3.X)
   WHERE Xset IS_A set OF integer_constant;
   (* Create the Y coordinates *)
   y_data[Xset] IS_A real;

   (* create the curve *)
   X_curve IS_A plt_curve(Xset,y_data,x_data);
   (* Make X_curve into the array for plt_plot_integer *)
   curves[curve_set] ALIASES (X_curve) WHERE
   curve_set IS_A set OF integer_constant;

METHOD default_self;
   RUN st.default_self;
   st.s1.L0 := 0.2{m}; (* make st more interesting *)
   RUN Plot_X.default_self;
   RUN X_curve.default_self;
   FOR i IN Xset DO
      y_data[i] := i;
   END FOR;
   X_curve.legend := ‘meter’;
   Plot_X.title := ‘Hook locations’;
   Plot_X.XLabel := ‘location’;
   Plot_X.YLabel := ‘hook #’;
END default_self;
END plot_spring_test;
```

6.4 RESULTING POSITION PLOT

We can compile the plot model and obtain the graph in with the following short script.

```
READ FILE force1d.a4c;
COMPILE pst OF plot_spring_test;
BROWSE {pst};
RUN {pst.st.reset};
SOLVE {pst.st} WITH QRSlv;
PLOT {pst.Plot_X} ;
SHOW LAST;
```

We can also obtain the plot by moving to pst.Plot_X in the Browser window and then pushing the Display->Plot button or then typing “Alt-d p”. We see the hooks are positioned near 0, 230, and 370 mm. We also see that xgraph sometimes makes less than pretty graphs.

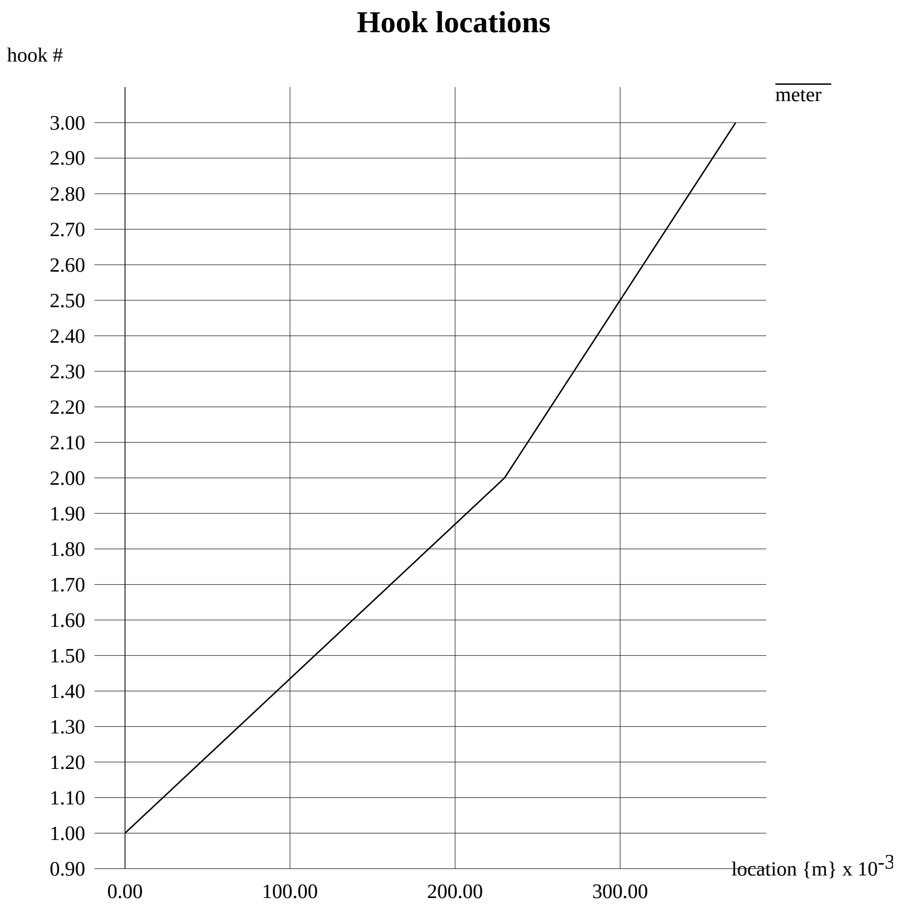

Figure 6-2 Plot_X in plot_spring_test

6.5 1-D MECHANICAL HOOK, SPRING, MASS, ANCHOR, AND FINGER MODELS

The models used in this chapter are very simple versions of masses and springs horizontally at rest, but possibly under tension, stretched between an anchor and a finger. Only the code absolutely necessary for this example is given here; the full code with methods and additional comments is given in force1d.a4c, an ASCEND modeling example in the library.

These models could easily be extended to include mass, momentum, and acceleration in two or three dimensions. Most of the methods in the force1d.a4c models are unedited from the code generated by the ASCEND Library button Edit->Suggest method. If you improve on these models, please share them with us and the rest of the ASCEND community.

```
REQUIRE “atoms.a4l”;
CONSTANT spring_constant REFINES real_constant DIMENSION M/T^2;
CONSTANT position_constant REFINES real_constant DIMENSION L;
ATOM position REFINES distance DEFAULT 0{m};
END position;

MODEL hook;
   F_left, F_right IS_A force;
   F_left = F_right;
   X IS_A position;
METHODS
METHOD default_self;
   (* ATOM defaults are fine *)
END default_self;
METHOD specify;
   F_right.fixed := TRUE;
END specify;
METHOD specify_float;
END specify_float;
END hook;

MODEL massless_spring(
   k IS_A spring_constant;
   h_left WILL_BE hook;
   h_right WILL_BE hook;
) WHERE (
   h_left, h_right WILL_NOT_BE_THE_SAME;
);
```

```
   L0, dx IS_A distance;
   h_right.X = h_left.X + L0 + dx;
   F = k * dx;
   h_left.F_right = F;
   h_right.F_left = F;
   F IS_A force;
METHODS
METHOD default_self;
   dx := 1{cm};
   L0 := 10{cm};
END default_self;
METHOD specify;
   L0.fixed := TRUE;
   RUN h_left.reset;
   RUN h_right.reset;
   h_left.F_right.fixed := FALSE;
   h_left.X.fixed := TRUE;
END specify;
METHOD specify_float;
   L0.fixed := TRUE;
   RUN h_left.specify_float;
   RUN h_right.specify_float;
END specify_float;
END massless_spring;

MODEL massless_block(
   h_left WILL_BE hook;
   h_right WILL_BE hook;
) WHERE (
   h_left, h_right WILL_NOT_BE_THE_SAME;
);
   width IS_A distance;
   h_left.F_right = h_right.F_left;
   h_right.X = h_left.X + width;
   X “center of the block” IS_A position;
   X = width/2 +  h_left.X;
METHODS
METHOD default_self;
   width := 3{cm};
END default_self;
METHOD specify;
   width.fixed := TRUE;
   RUN h_left.reset;
   h_left.F_right.fixed := FALSE;
   h_left.X.fixed := TRUE;
   RUN h_right.reset;
```

```
END specify;
METHOD specify_float;
   width.fixed := TRUE;
   RUN h_left.specify_float;
   RUN h_right.specify_float;
END specify_float;
END massless_block;

MODEL anchor(
   x IS_A position_constant;
   h_right WILL_BE hook;
);
   h_right.X = x;
   F = h_right.F_left;
   F IS_A force;
METHODS
METHOD default_self;
END default_self;
METHOD specify;
   RUN h_right.reset;
END specify;
METHOD specify_float;
END specify_float;
END anchor;

MODEL finger(
   h1 WILL_BE hook;
);
   pull IS_A force;
   h1.F_right = pull;
METHODS
METHOD default_self;
   pull := 3{N};
END default_self;
END finger;

MODEL finger_test;
NOTES ‘ascii-picture’ SELF {
                        ___     __
\\--O--/\/\/\/\/\/--O--|   |--O(_ \
                       |___|     \ \
(reference)-h1-(s1)-h2-(m1)-h3-(pinky)
}
END NOTES;
   reference IS_A anchor(0.0{m},h1);
   h1 IS_A hook;
   s1 IS_A massless_spring(100{kg/s^2},h1,h2);
   h2 IS_A hook;
   m1 IS_A massless_block(h2,h3);
   h3 IS_A hook;
   pinky IS_A finger(h3);
METHODS
METHOD default_self;
   RUN h1.default_self;
   RUN h2.default_self;
   RUN h3.default_self;
   RUN m1.default_self;
   RUN pinky.default_self;
   RUN reference.default_self;
   RUN s1.default_self;
END default_self;
METHOD specify;
   RUN m1.specify_float;
   RUN pinky.reset;
   RUN reference.specify_float;
   RUN s1.specify_float;
END specify;
END finger_test;

MODEL spring_test;
NOTES ‘ascii-picture’ SELF {
\\--O--/\/\/\/\/\/--O--\/\/\--O(\
(reference)-h1-(s1)-h2-(s2)-h3-(pinky)
}
END NOTES;
   reference IS_A anchor(0.0{m},h1);
   h1 IS_A hook;
   s1 IS_A massless_spring(100{kg/s^2},h1,h2);
   h2 IS_A hook;
   s2 IS_A massless_spring(75{kg/s^2},h2,h3);
   h3 IS_A hook;
   pinky IS_A finger(h3);
METHODS
METHOD default_self;
   RUN h1.default_self;
   RUN h2.default_self;
   RUN h3.default_self;
   RUN s2.default_self;
   RUN pinky.default_self;
   RUN reference.default_self;
   RUN s1.default_self;
END default_self;
```

```
METHOD specify;
   RUN pinky.reset;
   RUN reference.specify_float;
   RUN s1.specify_float;
   RUN s2.specify_float;
END specify;
END spring_test;

REQUIRE “plot.a4l”;
MODEL plot_spring_test;

   (* create our model *)
   st IS_A spring_test;

   (* Now gather the sampled data into an array for plotting *)
   x_data[Xset] ALIASES (st.h1.X,st.h2.X,st.h3.X)
   WHERE Xset IS_A set OF integer_constant;

   (* Now create the Y coordinates of the plot since there is no
    * natural Y coordinate in our MODEL.
    *)
   y_data[Xset] IS_A real; (* all will be assigned to 1.0 *)

   X_curve IS_A plt_curve(Xset,y_data,x_data);

   (* Make X_curve into the expected array for plt_plot *)
   curves[curve_set] ALIASES (X_curve) WHERE
   curve_set IS_A set OF integer_constant;

   Plot_X IS_A plt_plot_integer(curve_set,curves);
METHODS
METHOD default_self;
   RUN st.default_self;
   st.s1.L0 := 0.2{m};
   RUN X_curve.default_self;
   RUN Plot_X.default_self;
   FOR i IN Xset DO
      y_data[i] := i;
   END FOR;
   X_curve.legend := ‘meter’;
   Plot_X.title := ‘Hook locations’;
   Plot_X.XLabel := ‘location {m}’;
   Plot_X.YLabel := ‘hook #’;
END default_self;
END plot_spring_test;
```

CHAPTER 7 HOW TO DEFINE VARIABLES AND SCALING VALUES IN AN ASCEND MODEL

the purpose of this chapter

By now you have probably read Section 2, ”A Detailed ASCEND Example for Beginners: the modeling of a vessel,” on page 5 and seen an example of how to create a model using existing variable types in ASCEND. You found that variables of types area, length, mass, mass_density, and volume were needed and that they could be found in the library atoms.a4l. You want to know how to generalize on that; how to use variables, constants, and scaling values in your own models so that the models will be easier to solve.

This chapter is meant to explain the following things:

- The “Big Picture” of how variables, constants, and scaling values relate to the rest of the ASCEND IV language and to equations in particular. We’ll keep it simple here. More precise explanations for the language purist can be found in “The ASCEND IV language syntax and semantics” (syntax.pdf). You do not need to read about the “Big Picture” in order to read and use the other parts of this chapter, but you may find it helpful if you are having trouble writing an equation so that ASCEND will accept it.
- How to find the type of variable (or constant) you want. We keep a mess of interesting ATOM and CONSTANT definitions in atoms.a4l. We provide tools to search in already loaded libraries to locate the type you need.
- How to define a new type of variable when we do not have a predefined ATOM or CONSTANT that suits your needs. It is very easy to define your own variable types by copying code into an atoms library of your own from atoms.a4l and then editing the copied definition.
- How to define a scaling variable to make your equations much easier to solve.

7.1 THE BIG PICTURE: A TAXONOMY

As you read in Section 3, ”Preparing a model for reuse,” on page 25, simulations are built from MODEL and ATOM definitions, and MODEL and ATOM definitions are defined by creating types in an ASCEND language text file that you load into the ASCEND system. Figure 7-1 shows the types of objects that can be defined. You can see

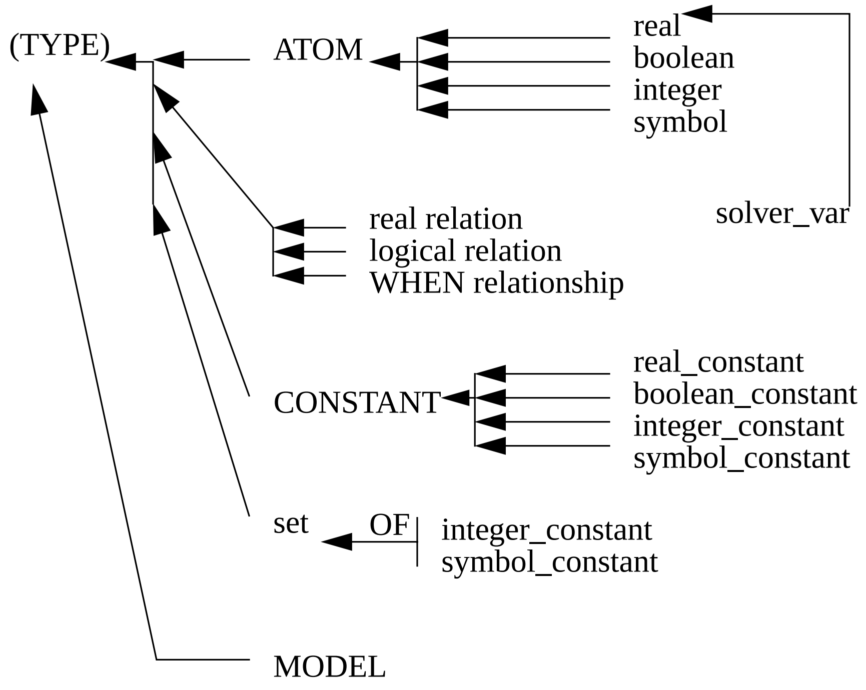

Figure 7-1 The Big Picture: How to think about variables

there are many more types than simply real variables used for writing equations. Some of these types can also be used in equations. You also see that there are three kinds of equations, not simply real relations. Throughout our documentation we call real relations simply “relations” because that is the kind of equation most people are interested in most of the time. Notice that “scaling values” do not appear in this diagram. We will cover scaling values at the end of this The major features of this diagram are:

ATOM

- Any variable quantity for use in relations, logical relations, or when statements or other computations. These come in the usual programming language flavors real, boolean, symbol, integer. Not all kinds of atoms can be used in all kinds of equations, as we shall explain when describing relations in a little bit. Atoms may be assigned values many times interactively, with the Script ASSIGN statement, with the METHOD := assignment operator, or by an ASCEND client such as a solver.

An ATOM may have attributes other than its value, such as .fixed in solver_var, but these attributes are not atoms. They are subatomic particles and cannot be used in equations. These attributes are interpretable by ASCEND clients, and assignable by the user in the same ways that the user assigns atom values.

Each subatomic particle instance belongs to exactly one atom instance (one variable in your compiled simulation). This contrasts with an atom instance which can be shared among several models by passing the atom instance from one model into another or by creating aliases for it.

CONSTANT

- Constants are "variables" that can be assigned no more than once. By convention, all constant types in atoms.a4l have names that end in _constant so that they are not easily confused with atoms. A constant gets a values from the DEFAULT portion of its type definition, by an interactive assignment, or by an assignment in the a model which uses the :== assignment operator. Constants cannot be assigned in a METHOD, nor can they be assigned with the := operator.

Integer and symbol constants can appear as members of sets or as subscripts of arrays. Integer, boolean, and symbol constants can be used to control SELECT statements which determine your simulation's structure at compile-time or to control SWITCH and WHEN behavior during problem solving .

set

- Sets are unordered lists of either integer or symbol constants. A set is assigned its value exactly once. The user interface always presents a set in sorted order, but this is for convenience only. Sets are useful for defining an array range or for writing indexed relations. More about sets and their use can be found in syntax.pdf.

relationships

- Relations and logical relations allow you to state equalities and inequalities among the variables and constants in you models. WHEN statements allow you to state relationships among the models and equations which depend on the values of variables in those models. Sets and symbols are not allowed in real or logical relations except when used as array subscripts.

Real relations relate the values of real atoms, real constants, and integer constants. Real relations cannot contain boolean constants and atoms, nor can they contain integer atoms.

Logical relations relate the values of boolean atoms and boolean constants. The SATISFIED operator makes it possible to include real relations in a logical relation. Neither integer atoms and constants nor real atoms and constants are allowed in logical relations. If you find yourself trying to write an equation with integer atoms, you are really creating a conditional model for which you should use the WHEN statement instead. See Section 13, ”Creating Conditional Models in Ascend,” on page 131 to learn about how ASCEND represents this kind of mathematical model. There are also a real variable types, *solver_integer* and *solver_binary*, which are used to formulate equations when the solver is expected to initially treat the variable as a real value but drive it to an integer or 0-1 value at the solution. The integer programming features of ASCEND are described in a technical report by Craig Schmidt not yet available electronically. See system.a4l for elementary details.

Like atoms, real and logical relations may have attributes, subatomic particles for use by ASCEND clients and users. The name of a relation can be used in writing logical relations and WHEN statements.

WHEN statements are outside the scope of this chapter; please see Section 13, ”Creating Conditional Models in Ascend,” on page 131 or syntax.pdf for the details.

MODEL

- A model is simply a container for a collection of atoms, constants, sets, relations, logical relations, when statements, and arrays of any of these. The container also specifies some of the methods that can be used to manipulate its contents. Compiling a model creates an instance of it-- a simulation.

SOLVER_VAR

- The real atom type *solver_var* is the type from which all real variables that you want the system to solve for must spring. If you define a real variable using a type which is not a refinement of solver_var, all solvers will treat that variable as an a scaling value or other given constant rather than as a variable.

Solver_vars have a number of subatomic attributes (upper_bound, lower_bound, and so forth) that help solvers find the solution of your model. ATOM definitions specify appropriate default values for these attributes that depend on the expected applications of the atom. These attribute values can (and should) be modified by methods in the final application model where the most accurate problem information is available.

Scaling value

- A real which is not a member of the *solver_var* family is ignored by the solver. Numerical solvers for problems with many equations in many variables work better if the error computed for each equation (before the system is solved) is of approximately size 1.0. This is most critical when you are starting to solve a new problem at values far, far away from the solution. When the error of one equation is much larger than the errors in the others, that error will skew the behavior of most numerical solvers and will cause poor performance.

This is one of the many reasons that scientists and engineers work with dimensionless models: the process of scaling the equations into dimensionless form has the effect of making the error of each equation roughly the same size even far away from the solution. It is sometimes easiest to obtain a dimensionless equation by writing the equation in its dimensional form using natural variables and then dividing both sides by an appropriate scaling value. We will see how to define an atom for scaling purposes in the last part of this chapter.

7.2 HOW TO FIND THE RIGHT VARIABLE TYPE

The type of real atom you want to use depends first on the dimensionality (length, mass/time, etc.) needed and then on the application in which the atom is going to be used. For example, if you are modeling a moving car and you want an atom type to describe the car's speed, then you need to find an atom with dimensionality length/time or in ASCEND terms L/T. There may be two or three types with this dimensionality, possibly including real_constants, a real scaling value, and an atom derived from solver_var.

Load atoms.a4l

The first step to finding the variable type needed is to make sure that atoms.a4l is loaded in your Library window from ascend4/models/atoms.a4l.

Find an ATOM or CONSTANT by units

The next step is to open the "ATOM by units" dialog found in the Library window's Find menu. This dialog asks for the units of the real variable type you want. For our example, speed, you would enter "feet/second," "furlongs/fortnight," "meter^3/second/ft^2" or any other combination of units that corresponds to the dimensionality L/T.

If the system is able to deduce the dimensionality of the units you have entered, it will return a list of all the currently loaded ATOM and CONSTANT definitions with matching dimensions. It may fail to understand the units, in which case you should try the corresponding SI units. If it understands the units but there are no matching atoms or constants, you will be duly informed. If there is no atom that meets your needs, you should create one as outlined in Section 7.3.

Selecting the right type

The resulting list of types includes a Code button which will display the definition of any of the types listed once you select (highlight) that type with the mouse. Usually you will need to examine several of the alternatives to see which one is most appropriate to the physics and mathematics of your problem. Compare the default, bounds, and nominal values defined to those you need. Check whether the type you are looking at is a CONSTANT or an ATOM.

You now know the name of the variable type you need, or you know that you must create a new one to suit your needs.

7.3 HOW TO DEFINE A NEW TYPE OF VARIABLE

In this section we will give examples of defining the atom and constant types as well as outline a few exceptional situations when you should NOT define a new type. More examples can be found and copied from atoms.a4l. You should define your new atoms in your personal atoms library.

Saving customized variable types

The normal location for this personal library is in the user data file ~\ascdata\myatoms.a4l. This file contains the following three lines and then the ATOM and CONSTANT definitions you create.

```
REQUIRE “atoms.a4l”; (* loads our atoms first *)
PROVIDE “myatoms.a4l”; (* registers your library *)
(* Custom atoms created by <insert your name here> *)
```

If you develop an interesting set of atoms for some problem domain outside chemical engineering thermodynamics, please consider mailing it to us through our web page.

The user data directory ~/ascdata may have a different name if you are running under Windows and do not have the environment variable HOME defined. It may be something like C:\ascdata or \WINNT\Profiles\Your Name\ascdata. When ASCEND is started, it prints out the name of this directory.

When you write a MODEL which depends on the definition of your new atoms, do not forget to add the statement

```
REQUIRE “myatoms.a4l”;
```

at the very top of your model file so that your atoms will be loaded before your model definitions try to use them.

7.3.1 A NEW REAL VARIABLE FOR SOLVER USE

Suppose you need an atom with units {dollar/ft^2/year} for some equation relating amortized construction costs to building size. Maybe this example is a bit far fetched, but it is a safe bet that our library is not going to have an atom or a constant for these units. Here is the standard incantation for defining a new variable type based on solver_var. ASCEND allows a few permutations on this incantation, but they are of no practical value. The parts of this incantation that are in *italics* should be changed to match your needs. You can skip the comments, but you **must** include the units of the default on the bounds and nominal.

```
ATOM amortized_area_cost
REFINES solver_var DEFAULT 3.0 {dollar/ft^2/year};
   lower_bound := 0 {dollar/ft^2/year};
   (* minimum value *)
   upper_bound := 10000 {dollar/ft^2/year};
   (* maximum value for any sane application *)
   nominal := 10 {dollar/ft^2/year};
   (* expected size for all reasonable applications*)
END amortized_area_cost;
```

In picking the name of your atom, remember that names should be as self-explanatory as possible. Also avoid choosing a name that ends in *_constant* (as this is conventionally applied only to CONSTANT definitions) or *_parameter*. Parameter is an extremely ambiguous and therefore useless word. Also remember that the role a variable plays in solving a set of equations depends on how the solver being applied interprets *.fixed* and other attributes of the variable.

Exceptions

If an atom type matches all but one of the attributes you need for your problem, say for example the upper_bound is way too high, use the existing variable type and reassign the bound to a more sensible value in the *default_self* method of the model where the variable is created. Having a dozen atoms defined for the same units gets confusing in short order to anyone you might share your models with.

The exception to the exception (yes, there always seems to be one of those) is the case of a lower_bound set at zero. Usually a lower_bound of zero indicates that there is something inherently positive about variables of that type. Variables with a bound of this type should not have these physical bounds expanded in an application. Another example of this type of bound is the upper_bound 1.0 on the type *fraction*.

For example, negative temperature just is not sensible for most physical systems. ASCEND defines a *temperature* atom for use in equations involving the absolute temperature. On the other hand, a temperature difference, delta T, is frequently negative so a separate atom is defined. Anyone receiving a model written using the two types of atoms which both have units of {Kelvin} can easily tell which variables might legitimately take on negative values by noting whether the variable is defined as a *temperature* or a *delta_temperature*.

7.3.2 A NEW REAL CONSTANT TYPE

Real constants which do not have a default value are usually needed only in libraries of reusable models, such as components.a4l, where the values depend on the end-user's selection from alternatives in a database. The standard incantation to define a new real constant type is:

```
CONSTANT critical_pressure_constant
REFINES real_constant DIMENSION M/L/T^2;
```

Here again, the *italic* parts of this incantation should be redefined for your purpose.

Universal exceptions and unit conversions

It is wasteful to define a CONSTANT type and a compiled object to represent a *universal* constant. For example, the thermodynamic gas constant, R = 8.314... {J/mole/K}, is frequently needed in modeling chemical systems. The SI value of R does not vary with its application. Neither does the value of π. Numeric constants of this sort are better represented as a numeric coefficient and an appropriately defined unit conversion. Consider the ideal gas law, PV = NRT and the ASCEND unit conversion {GAS_C} which appears in the library ascend4/models/measures.a4l. This equation should be written:

```
   P * V = n * 1.0{GAS_C} * T;
```

Similarly, area = pi*r^2 should be written

```
   area = 1{PI} * r^2;
```

The coefficient 1 of {GAS_C} and {PI} in these equations takes of the dimensionality of and is multiplied by the conversion factor implied by the UNITS definition for the units. If we check measures.a4l, we find the definition of PI is simply {3.14159...} and the definition of GAS_C is {8.314... J/mole/K} as we ought to expect.

For historical reasons there are a few universal constant definitions in atoms.a4l. New modelers should not use them; they are only provided to support outdated models that no one has yet taken the time to update.

7.3.3 NEW TYPES FOR INTEGERS, SYMBOLS, AND BOOLEANS

The syntax for ATOM and CONSTANT definitions of the non-real types is the same as for real number types, except that units are not involved. Take your best guess based on the examples above, and you will get it right. If even that is too hard, more details are given in syntax.pdf.

7.4 HOW TO DEFINE A SCALING VARIABLE

A scaling variable ATOM is defined with a name that ends in *_scale* as follows. Note that this ATOM does not refine solver_var, so solvers will not try to change variables of this type during the solution process.

```
ATOM distance_scale REFINES real DEFAULT 1.0{meter};
END distance_scale;
```

ASCEND cannot do it all for you

ASCEND uses a combination of symbolic and numerical techniques to create and solve mathematical problems. Once you get the problem close to the solution, ASCEND can internally compute its own scaling values for relations, known elsewhere as “relation nominals”, assuming you have set good values for the *.nominal* attribute of all the variables. It does this by computing the largest additive term in each equation. The absolute value of this term is a very good scaling value.

This internal scaling works quite well, but not when the problem is very far away from the solution so that the largest additive terms computed are not at all representative of the physical situation being modeled. The *scale_self* method, which should be written for every model as described in Section 10.4.4, should set the equation scaling values you have defined in a MODEL based on the best available information. In a chemical engineering flowsheeting problem, for example, information about a key process material flow should be propagated throughout the process flowsheet to scale all the other flows, material balance equations, and energy balance equations.

Scaling atom default value

The default value for any scaling atom should always be 1.0 in appropriate SI units, so that the scaling will have no effect until you assign a problem specific value. Multiplying or dividing both sides of an equation by 1.0 obviously will not change the mathematical behavior, but you do not want to change the behavior arbitrarily either--you want to change it based on problem information that is not contained in your myatoms.a4l file.

CHAPTER 8 ENTERING DIMENSIONAL EQUATIONS FROM HANDBOOKS

Often in creating an ASCEND model one needs to enter a correlation given in a handbook that is written in terms of variables expressed in specific units. In this chapter, we examine how to do this easily and correctly in a system like ASCEND where all equations must be dimensionally correct.

8.1 EXAMPLE 1— VAPOR PRESSURE

Our first example is the equation to express vapor pressure using an Antoine-like equation of the form:

$$\ln(P^{sat}) = A - \frac{B}{T + C} \quad \textbf{(8.1)}$$

where P^{sat} is in {atm} and T in {R}. When one encounters this equation in a handbook, one then finds tabulated values for A, B and C for different chemical species. The question we are addressing is:

How should the modeler enter this equation into ASCEND so he or she can then enter the constants A, B, and C with the exact values given in the handbook?

ASCEND uses SI units internally. Therefore, P would have the units {kg/m/s^2}, and T would have the units {K}.

Equation 8.1 is, in fact, dimensionally incorrect as written. We know how to use it, but ASCEND does not as ASCEND requires that we write dimensionally correct equations. For one thing, we can legitimately take the natural log *(ln)* only of unitless quantities. Also, the handbook will tabulate the values for A, B and C without units. If A is dimensionless, then B and C would require the dimensions of temperature.

The mindset to enter such equations is to make all quantities that must be expressed in particular units into dimensionless quantities which have the correct numerical value.

We illustrate in the following subsections just how to do this conversion. It proves to be very straight forward to do.

8.1.1 CONVERTING THE *LN* TERM

Convert the quantity within the *ln()* term into a dimensionless number that has the value of pressure when pressure is expressed in {atm}.

Very simply, we write

```
P_atm = P/1{atm};
```

Note that P_atm has to be a *dimensionless* quantity here.

We then rewrite the LHS of Equation 8.1 as

```
ln(P_atm)
```

Suppose P = 2 {atm}. In SI units P= 202,650 {kg/m/s^2}. In SI units, the dimensional constant 1{atm} is about 101,325 {kg/m/s^2}. Using this definition, P_atm has the value 2 and is dimensionless. ASCEND will not complain with P_atm as the argument of the *ln* () function, as it can take the natural log of the dimensionless quantity 2 without any difficulty.

8.1.2 CONVERTING THE RHS

We next convert the RHS of Equation 8.1, and it is equally as simple. Again, convert the temperature used in the RHS into:

```
T_R = T/1{R};
```

ASCEND converts the dimensional constant 1{R} into 0.55555555...{K}. Thus T_R is dimensionless but has the value that T would have if expressed in {R}.

8.1.3 IN SUMMARY FOR EXAMPLE 1

We do not need to introduce the intermediate dimensionless variables. Rather we can write:

```
ln(P/1{atm}) = A - B/(T/1{R} + C);
```

as a correct form for the dimensional equation. When we do it in this way, we can enter A, B and C as dimensionless quantities with the values exactly as tabulated.

8.2 FAHRENHEIT— VARIABLES WITH OFFSET

What if we write Equation 8.1 but the handbook says that T is in degrees Fahrenheit, i.e., in {F}? The conversion from {K} to {F} is

T{F} = T{K}*1.8 - 459.67

and the 459.67 is an *offset*. ASCEND does not support an offset for units conversion. We shall discuss the reasons for this apparent limitation in Section 8.4.

You can readily handle temperatures in {F} if you again think as we did above. The rule, even for units requiring an offset for conversion, remains: convert a dimensional variable into dimensionless one such that the dimensionless one has the proper value.

Define a new variable

```
T_degF = T/1{R} - 459.67;
```

Then code Equation 8.1 as

```
ln(P/1{atm}) = A - B/(T_degF + C);
```

when entering it into ASCEND. You will then enter constants A, B, and C as dimensionless quantities having the values exactly as tabulated. In this example we *must* create the intermediate variable T_degF.

8.3 EXAMPLE 3— PRESSURE DROP

From the Chemical Engineering Handbook by Perry and Chilton, Fifth Edition, McGraw-Hill, p10-33, we find the following correlation:

$$\Delta P'_a = \frac{y(V_g - V_l)G^2}{144g} \qquad \textbf{(8.2)}$$

where the pressure drop on the LHS is in psi, y is the fraction vapor by weight (i.e., dimensionless), Vg and V_l are the specific volumes of gas and liquid respectively in ft^3/lbm, G is the mass velocity in lbm/hr/ft^2 and g is the acceleration by gravity and equal to 4.18x10^8 ft/hr^2.

We proceed by making each term dimensionless and with the right numerical value for the units in which it is to be expressed. The following is the result. We do this by simply dividing each dimensional variable by the correct unit conversion factor.

```
delPa/1{psi} = y*(Vg-Vl)/1{ft^3/lbm}*
               (G/1{lbm/hr/ft^2})^2/(144*4.18e8);
```

8.4 THE DIFFICULTY OF HANDLING UNIT CONVERSIONS DEFINED WITH OFFSET

How do you convert temperature from Kelvin to centigrade? The ASCEND compiler encounters the following ASCEND statement:

```
d1T1 = d1T2 + a.Td[4];
```

and d1T1 is supposed to be reported in centigrade. We know that ASCEND stores termperatures in Kelvin {K}. We also know that one converts {K} to {C} with the following relationship

T{C} = T{K} - 273.15.

Now suppose d1T2 has the value 173.15 {K} and a.Td{4} has the value 500 {K}. What is d1T1 in {C}? It would appear to have the value 173.15+500-273.15 = 400 {C}. But what if the three variables here are really temperature differences? Then the conversion should be

T{dC} = T{dK}.

where we use the notation {dC} to be the units for temperature difference in centigrade and {dK} for differences in Kelvin. Then the correct answer is 173.15+500=673.15 {dC}.

Suppose d1T2 is a temperature and d1T2 is a temperature difference (which would indicate an unfortunate but allowable naming scheme by the creator of this statement). It turns out that a.Td[4] is then required to be a temperature and not a temperature difference for this equation to make sense. We discover that an equation that involves the sums and differences of temperature and temperature difference variables will have to have an equal number of positive and negative temperatures in it to make sense, with the remaining having to be temperature differences. Of course if the equation is a correlation, such may not be the case, as the person deriving the correlation is free to create an equation that “fits” the data without requiring the equation be dimensionally (and physically) reasonable.

We could create the above discussion just as easily in terms of pressure where we distinguish absolute from gauge pressures (e.g., {psia} vs. {psig}). We would find the need to introduce units {dpisa} and {dpsig} also.

8.4.1 GENERAL OFFSET AND DIFFERENCE UNITS

Unfortunately, we find we have to think much more generally than the above. Any unit conversion can be introduced both with and without offset. Suppose we have an equation which involves the sums and diffences of terms t1 to t4:

t1 + t2 - (t3 + t4) = 0 **(8.3)**

where the units for each term is some combination of basic units, e.g., {ft/s^2/R}. Let us call this combination {X} and add it to our set of allowable units, i.e., we define

{X} = {ft/s^2/R}.

Suppose we define units {Xoffset} to satisfy:

{Xoffset} = {X} - 10

as another set of units for our system. We will also have to introduce the concept of {dX} and and should probably introduce also {dXoffset} to our system, with these two obeying

{dXoffset} = {Xoffset}.

For what we might call a “well-posed” equation, we can argue that the coefficient of variables in units such as {Xoffset} have to add to zero with the remaining being in units of {dX} and {dXoffset}. Unfortunately, the authors of correlation equations are not forced to follow any such rule, so you can find many published correlations which make the most awful (and often unstated) assumptions about the units of the variables being correlated.

Will the typical modeler get this right? We suspect not. We would need a very large number of unit conversion combinations in both absolute, offset and relative units to accomodate this approach.

We suggest that our approach to use only absolute units with no offset is the least confusing for a user. Units conversion is then just multiplication by a factor both for absolute {X} and difference {dX} units— we do not have to introduce difference variables because we do not introduce offset units.

When users want offset units such as gauge pressure or Fahrenheit for temperature, they can use the conversion to dimensionless variables having the right value, using the style we introduced above, i.e.,

T_defF = T/1{R} - 459.67

and

P_psig = P/1{psi} - 14.696

as needed.

Both approaches to handling offset introduce undesirable and desirable characteristics to a modeling system. Neither allow the user to use units without thinking carefully. We voted for this form because of its much lower complexity.

CHAPTER 9 DEFINING NEW UNITS OF MEASURE

Occasionally units of measure are needed that do not come predefined in the ASCEND system. You can define a new unit of measure by defining the conversion factor. In this chapter, we examine how to do this easily for an individual user and on a system-wide basis.

9.1 *CAVEATS*

Order matters!

Order matters for defining units of measure in three ways.

- a unit of measure must be defined before it is used anywhere.
- the first definition ASCEND reads for a unit of measure is the only definition ASCEND sees.
- new units can be defined only from already defined units.

Measuring units are absolutely global in the ASCEND environment—they are not deleted when the Library of types is deleted. Once you define a unit's conversion factor, you are stuck with it until you shut down and restart ASCEND. For any unit conversion definition, only the first conversion factor seen is accepted. Redefinitions of the same unit are ignored.

Multiplicative unit conversions only!

The various units ASCEND uses are all obtained by conversion factors (multiplication only) from the SI units. So, for example, temperatures may be in degrees Rankine but not in Fahrenheit. In this chapter we address creating new conversion factors. For handling non-multiplicative conversions (such as the Fahrenheit or Celsius offsets) see Section 8.2.

9.2 INDIVIDUALIZED UNITS

There are two scenarios for individualized units of measure. One in which you need a measure defined only for a specific model and another in which you want to define a measure that you will use throughout your modeling activities in the future. The syntax for both is the same, but where best to put the UNITS statement differs.

9.2.1 UNITS OF MEASURE FOR A SPECIFIC MODEL

Units of measure which are used in only one model can be defined at the beginning of the model itself or before the model, but not the units appear in the model definition. Let us suppose you want to measure speed in {furlong/fortnight} in a model. ASCEND does not define furlong, fortnight, or furlong/fortnight. (We cannot find standard definitions for them!).

```
MODEL mock_turtle;
   d IS_A distance;
   delta_t IS_A time;
   s IS_A speed
   s = d/delta_t;
   (* We really should write s * delta_t = d;
    * to avoid division by zero.
    *)
UNITS
   furlong = {3.17*kilometer};
   fortnight = {10*day};
END UNITS;
METHODS
METHOD default_self;
   d := 1 {furlong};
   t := 5 {hours};
END default_self;
(* other standard methods omitted *)
END mock_turtle;
```

In mock_turtle we define *furlong* and *fortnight* conversions before they are used in the methods and before any equations which use them. Also, notice that even though ASCEND rejects this model `mock_turtle`, as it will because of the missing “;” after “`speed`” in the fourth line, *furlong* and *fortnight* still get defined. The UNITS statement can appear in any context and gets processed regardless of any other errors in that context.

9.2.2 UNITS OF MEASURE FOR ALL YOUR PERSONAL MODELS

If you commonly use a set of units that is not in the default ASCEND library measures.a4l, you can create your own personal library of units in the user data directory ascdata. The location of this directory is given by ASCEND at the end of all the start-up spew it prints to the Console window (or xterm under UNIX) as shown below. You will see a path other than `/usr0/ballan/` of course.

```
-----------------------------------
User data directory is /usr0/ballan/ascdata
-----------------------------------
```

Create the library file myunits.a4l in your ascdata directory. This file should contain a UNITS statement and any comments or NOTES you wish to make. This file should contain any conversions that you change often. For example:

```
UNITS (* Units for Norway, maybe?*)
euro = {1*currency};
(* currency is the fundamental financial unit *)
kroner = {0.00314*euro};
nk = {kroner};
USdollar = {0.9*euro};
CANdollar = {0.65*USdollar};
END UNITS;
```

Note that this file contains a definition of USdollar different from that given in the standard library measures.a4l. ASCEND will warn you about the conflict. You must load myunits.a4l into ASCEND before atoms.a4l or any of our higher level libraries. You can ensure that this happens by putting the statement

```
REQUIRE “myunits.a4l”;
```

on the very first line in all your model definition files.

9.3 NEW SYSTEM-WIDE UNITS

Suppose you are maintaining ASCEND on a network of computers with many users. You have a standard set of models stored in a centrally located directory, and you want to define units for use by everyone on the network. In this case, just edit models/measures.a4l, the default units of measure library. ASCEND is an open system.

Make the new unit conversion definition statement(s) of the form

```
newunit = {combination of old units};
```

as described in Section 9.2. In the file measures.a4l, add your statement(s) anywhere inside the block of definitions that starts with UNITS and ends with “END UNITS.” The existing definitions are divided up into groups by comment statements. If your conversion belongs to one of the groups, it is best to put the conversion in that group. The groups are given in Table 9-1.

Table 9-1 Groups of units in the current measures library

distance
mass
time
molecular quantities
money
reciprocal time (frequency)
area
volume
force
pressure
energy
power
absolute viscosity
electric charge
miscellaneous electromagnetic
swiped from C math.h
constant based conversions
subtly dimensionless measures
light quantities
miscellaneous rates
time variant conversions

9.4 SEND THEM IN

We are always on the lookout for useful unit conversions to add to measures.a4l. If you create a myunits.a4l containing unit conversion definitions of general use (i.e. not currency exchange rates and other time-varying conversions), please mail us a copy and include your name in a comment. Thank you very much.

CHAPTER 10 HOW (AND WHY) TO WRITE STANDARD METHODS

In this chapter we describe a methodology (pun intended) which can help make anyone who can solve a quadratic equation a mathematical modeling expert. This methodology helps you to avoid mistakes and to find mistakes quickly when you make them. Finding bugs weeks after creating a model is annoying, inefficient, and (frequently) embarrassing. Because METHOD code can be large, we do not include many examples here. See Chapter 3, “Preparing a model for reuse,” on page 25 for detailed examples. One of the advantages of this methodology is that it allows almost automatic generation of methods for a model based on the declarative structure (defined parts and variables) in the model, as we shall see in Section 10.10. Even if you skip much of this chapter, read Section 10.10

We divide methods into _self and _all categories. The premise of this method design is that we can write the _self methods incrementally, building on the already tested methods of previous MODEL parts we are reusing. In this way we never have to write a single huge method that directly manipulates 100s of variables in a hierarchy. The _all methods are methods which simply “top off” the _self methods. With an _all method, you can treat just a *part* of a larger simulation already built as a self-contained simulation.

Usually discovery of the information you need to write the methods proceeds in the order that they appear below: check, default, specify, bound, scale.

10.1 WHY YOU SHOULD FOLLOW OUR WAYS

If debugging is the repair of modeling errors, then modeling must be the process of creating those errors.[1]

Some geniuses make more mistakes than anyone else -- because they try more things that anyone else. Part (perhaps a very large part) of what makes such a genius different from the rest of humanity is that they quickly recognize their own mistakes and move on to something else before anyone notices that they have screwed up! Solving a problem as far and as fast as you can, and then going back to criticize

1. Somebody famous said something like this about programming computers. The principle holds.

every aspect of the solution with an eye to improving it is how you usually discover right answers. Do it our way so that **ASCEND can help you find your mistakes.**

We (geniuses or not we'll leave to our users to decide) have found that it is best to write mathematical MODELs and mathematical modeling software in ways which make our mistakes (or your mistakes) very easy to detect. This way it is easier to find and fix problems early, instead of discovering the bug while the boss and the vice-president (or the advisor and the industrial sponsor) are hovering near. The ASCEND system will not force you to write standard methods in your models. METHODs of the sort we advocate here make your MODELs much easier to use and much more reliable. They pay off in the short run as well as the long run. These are *guidelines,* not *laws*: geniuses know when to color outside the lines.

If you do not write the standard methods, your MODEL will inherit the ones given in the library basemodel.a4l. The *ClearAll* and *reset* methods here will work for you if you follow the guidelines for the method *specify.* The other methods defined in basemodel.a4l (*check_self, default_self, bound_self, scale_self, check_all, default_all, bound_all, scale_all*) all contain STOP statements which will warn you that you have skipped something important, should you accidentally call one of these methods. If you create a model for someone else and they run into one of these STOP errors while using your model, that error is *your* fault.

10.2 METHODS *_SELF VS *_ALL

When you create a model definition, you create a container holding variables, equations, arrays, and other models. You create methods in the same definition to control the state of (the values stored in) all these parts. ASCEND lets you share objects among several models by passing objects through a model interface (the MODEL parameter list), by creating ALIASES for parts within contained objects, and even by merging parts (though this is a dumb idea for any object bigger than a variable).

Too many cooks spoil the soup.

The problem this creates for you as a METHOD writer is to decide which of the several MODELs that share an object is responsible for updating that variable's default, bounds, and nominal values. You could decide that every model which shares a variable is responsible for these values. This will lead to many, many, many hard to understand conflicts as different models all try to manage the same value. The sensible approach is to make only one model responsible for the bounding, scaling, and default setting of each variable: the model which creates the variable in the first place.

Use *_self methods on locally created variables and parts

Consider the following model and creating the *_self methods default_self, check_self, bound_self, and scale_self for it.

```
MODEL selfish(
   external_var WILL_BE solver_var;
   out_thingy WILL_BE input_part;
);
   my_variable IS_A solver_var;
   peek_at_variable ALIASES out_thingy.mabob.cost;
   my_thingy IS_A nother_part;
   navel_gaze ALIASES my_thingy.mabob.cost;
END selfish;
```

This model should manage the value of the variable it creates: *my_variable*. *External_var* comes in from the outside, so some other model will create and manage it. *Peek_at_variable* and *navel_gaze* also are not created here and should not be managed in the *_self methods of *selfish*.. We want to default, bound, or scale variables in complex parts we create also. We should call *my_thingy.default_self* whenever *default_self* is called for this model. We should not call *out_thingy.default_self*, however, as some other model will do so.

Use *_all methods to manage a troublesome part

Any mathematical subproblem in a large simulation may need to be isolated for debugging or solving purposes. When this is done using the Browser and Solver tools, you still need to call scaling, bounding, and checking methods for all parts of the isolated subproblem, even for those parts that came in from the outside. This is easily done by writing *_all methods. In the example above, *scale_all* will scale *external_var* and call *out_thingy.scale_all* because these parts are defined using WILL_BE statements. *scale_all* will then call the local*scale_self* to do all the normal scaling.

That’s the big picture of _self and _all methods. Each kind of method (bound, scale, default, check) has its own peculiarities which we cover in Section 10.4 and Section 10.5, but they all follow the rules above which distinguish among variables and parts defined with WILL_BE (managed in *_all only), IS_A (managed in *_self only), and ALIASES (not our responsibility).

10.3 HOW TO WRITE CLEARALL AND RESET

Writing these two standard methods in your model is very simple— do nothing. You may wish to write alternative reset_* methods as we shall discuss. These methods are inheritted by all models from the definitions in basemodel.a4l. Just so you know, here is what they do.

10.3.1 CLEARALL

This method finds any variable that is a solver_var or refinement of solver_var and changes the *.fixed* flag on that var to FALSE. This method does not change the value of *.included* flags on relations or return boolean, integer, or symbol variables to a default value.

10.3.2 RESET

This method calls *ClearAll* to bring the model to a standard state with all variables unfixed (free), then it calls the user-written *specify* method to bring the model with an equal number of variables to calculate and equations to solve. Normally you do not need to write this method: your models will inherit this one unless you override it (redefine it) in your MODEL.

This standard state is not necessarily the most useful starting state for any particular application. This method merely establishes a base case. There is no ‘one perfect “reset”’ for all purposes. Other reset_whatElseYouWant methods can also be written. The name of a method is a communication tool. Please use meaningful names as long as necessary to tell what the method does. Avoid cryptic abbreviations and hyper-specialized jargon known only to you and your three friends when you are naming methods; however, do not shy away from technical terms common to the engineering domain in which you are modeling.

10.4 THE *_SELF METHODS

The following methods should be redefined by each reusable library MODEL. Models that do not supply proper versions of these methods are usually very hard to reuse.

10.4.1 METHOD CHECK_SELF

This method should be written first, though it is run last. Just like they taught you in elementary school, always check your work. Start by defining criteria for a successful solution that will not be included in the equations solved and compute them in this method. As you develop your MODEL, you should expect to revise the check method from time to time, if you are learning anything about the MODEL. We frequently change our definition of success when modeling.

When a mathematical MODEL is solved, the assumptions that went into writing (deriving) the equations should be checked. Usually there are redundant equations available (more than one way to state the physics or economics mathematically). These should be used to check the particularly tricky bits of the MODEL.

Check that the physical or intuitive (qualitative) relationships among variables you expect to hold are TRUE, especially if you have not written such relationships in terms of inequalities (x*z <= y) in the MODEL equations.

In some models, checking the variable values against absolute physical limits (temperature > 0{K} and temperature < Tcritical for example) may be all that is necessary or possible. Do not check variable values against their .lower_bound or .upper_bound, as any decent algebraic solver or modeling system (e.g. ASCEND) will do this for you.

If a check fails, use a STOP statement to notify yourself (or you MODEL using customer) that the solution may be bogus. STOP raises an error signal and issues an error message. STOP normally also stops further execution of the method and returns control to a higher level, though there are interactive tools to force method execution to continue. STOP does not crash the ASCEND system.

10.4.2 METHOD DEFAULT_SELF

This method should set default values for any variables declared locally (IS_A) to the MODEL. It also should run *default_self* on *all* the complex parts that are declared locally (with IS_A) in the MODEL. If the atoms you use to define your variables have a suitable default already, then you do not need to assign them a default in this method.

This method should not run any methods on MODEL parts that come via WILL_BE in the definition's parameter list. This method also should not change the values of variables that are passed in through the parameter list. Sometimes there will be nothing for this method to do. Define it anyway, leaving it empty, so that any writer reusing this model as part of a higher level model can safely assume it is there and call it without having to know the details.

When a top-level simulation is built by the compiler, this method will be run (for the top-level model) at the end of compilation. If your model's *default_self* method does not call the lower level *default_self* methods in your model's IS_A'd parts, it is quite likely that your model will not solve.

10.4.3 METHOD BOUND_SELF

Much of the art of nonlinear physical modeling is in bounding the solution. This method should update the bounds on locally defined (IS_A) variables and IS_A defined MODEL parts. Updating bounds requires some care. For example, the bounds on fractions frequently don't need updating. This method should not bound variables passed into the MODEL definition or parts passed into the definition.

A common formula for updating bounds is to define a region around the current value of the variable. A linear region size formula, as an example, would be:

$$\text{x.bound} = x \pm \Delta \bullet \text{x.nominal}$$

or

```
v.upper_bound := v + boundwidth * v.nominal;
v.lower_bound := v - boundwidth v.nominal;
```

Care must be taken that such a formula does not move the bounds (particularly lower bounds) out so far as to allow non-physical solutions. Logarithmic bounding regions are also simple to calculate. Here boundwidth IS_A bound_width; boundwidth is a real atom (but not a solver_var) or some value you can use to determine how much "wiggle-room" you want to give a solver.

Small powers of 4 and 10 are usually good values of boundwidth. Too small a boundwidth can cut off the portion of number space where the solution is found. Too large a bound width can allow solvers to wander for great distances in uninteresting regions of the number space.

10.4.4 METHOD SCALE_SELF

Most nonlinear (and many linear) models cannot be solved without proper scaling of the variables. *scale_self* should reset the .nominal value on every real variable in need of scaling. It should then call the *scale_self* method on all the locally defined (IS_A) parts of the MODEL. 0.0 is the worst possible nominal value. A proper nominal is one such that you expect at the solution $0.1 \le abs\left(\frac{\mathrm{x}}{\mathrm{x.nominal}}\right) \le 10$. This method should not change the scaling of models and variables that are received through the parameter list of the MODEL.

Variables (like fractions) bounded such that they cannot be too far away from 1.0 in magnitude probably don’t need scaling most of the time if they are also bounded away from 0.0.

Some solvers, but not all, will attempt to scale the equations and variables by heuristic matrix-based methods. This works, but inconsistently; user-defined scaling is generally superior. ASCEND makes scaling equations easy to do. You scale the variables, which can only be done well by knowing something about where the solution is going to be found (by being an engineer, for example.) Then ASCEND can calculate an appropriate equation-scaling by efficient symbolic methods.

10.5 THE *_ALL METHODS

10.5.1 METHOD DEFAULT_ALL

This method assumes that the arguments to the MODEL instance have not been properly initialized, as is frequently the case in one-off modeling efforts. This method should run the *default_all* method on each of the parts received through the parameter list via WILL_BE statements and should give appropriate default values to any variables received through the parameter list. After these have been done, it should then call *default_self* to take care of all local defaults.

10.5.2 METHOD CHECK_ALL

When solving only a part of a simulation, it is necessary to check the models and variables passed into the part as well as the locally defined parts and variables. This method should call *check_all* on the parts received as WILL_BE parameters, then call *check_self* to check the locally defined parts and equations.

10.5.3 METHOD BOUND_ALL

This method should be like *bound_self* except that it bounds the passed in variables and calls *bound_all* on the passed in parts. It should then call *bound_self*.

10.5.4 METHOD SCALE_ALL

This method should be like *scale_self* above except that it scales the variables received through the parameter list and calls scale_all on the passed in parts. It should then call *scale_self* to take care of the local variables and models.

10.6 METHOD SPECIFY

Assuming ClearAll has been run on the MODEL, this method should get the MODEL to a condition called ‘square’: the case where there are as many variables with .fixed == FALSE as there equations eligible to compute them. This is one of the hardest tasks ever invented by mathematicians if you go about it in the wrong way. We think we know the right way.

Actually, ‘square’ is a bit trickier to achieve than simply counting equations and variables. Solvers, such as QRSlv in ASCEND, may help greatly with the bookkeeping.

The general approach is to:

1. Run “specify” for all the parts (both passed in and locally defined) that are not passed on into other parts.
2. Fix up (by tweaking .fixed flags on variables) any difficulties that arise when parts compete to calculate the same variable.
3. Use the remaining new local variables to take care of any leftover equations among the parts and any new equations written locally.

At all steps 1-3 pay special attention to indexed variables used in indexed equations. Frequently you must fix or free N or N-1 variables indexed over a set of size N, if there are N matching equations. In general, if you think you have *specify* correctly written, change the sizes of all the sets in your MODEL by one and then by two members. If your specify method still works, you are probably using sets correctly. Pursuing “symmetry,” the identical treatment of all variables defined in a single array, usually helps you write specify correctly.

When writing models that combine parts which do not share very well, or which both try to compute the same variable in different ways, it may even be necessary to write a WHEN statement to selectively TURN OFF the conflicting equations or MODEL fragments. An object or equation USEd in any WHEN statement is turned off by default and becomes a part of the solved MODEL only when the condition of some CASE which USEs that object is matched.

The setting of boolean, integer, and symbol variables which are controlling conditions of WHEN and SWITCH statements should be done in the specify method.

There is no ‘one perfect “specify”’ for all purposes. This routine should merely define a reasonably useful base configuration of the MODEL. Other specify_whatElseYouWant methods can also be written. Again, the name of a method is a communication tool. Please use meaningful names as long as necessary to tell what the method does. Avoid cryptic abbreviations and hyper-specialized jargon known only to you and your three friends when you are naming methods; however, do not shy away from technical terms common to the engineering domain in which you are modeling.

10.7 METHOD VALUES

In a final application MODEL, you should record at least one set of input values (values of the fixed variables and guesses of key solved-for variables) that leads to a good solution. Do this so no one need reinvent that set the next time you use the MODEL or someone picks the MODEL up after you.

10.8 METHODS AND CHEMICAL PROCESS MODELS

This next tip is due to Duncan Coffey. When creating a process model (such as a flash tank) which involves an equilibrium state calculation connected to input or output process flow streams, take care in ordering the calls to these stream and thermodynamic parts. Specifically, calls to methods in the equilibrium calculation should be done *after* calls to methods in the streams. For example in MODEL dyn_flash.a4l:detailed_tray:

```
METHOD default_all;
   Qin := 0 {watt};
   RUN vapin.default_self;
   RUN liqin.default_self;
   RUN vapout.default_self;
   RUN liqout.default_self;
   RUN state.default_self;
   RUN default_self;
END default_all;
```

Here we see *state.default self* is called last. The part state shares information with vapout and liqout, naturally.

10.9 SUMMARY

adding our *standard* methods to a model definition

We have defined a set of standard methods for ASCEND models which we insist a modeler provide before we will allow a model to be placed in any of our model libraries. These are listed in Table 10-1. As should

Table 10-1 List of standard methods we insist be added for each of the types in our ASCEND library of type definitions

method	**description**
default_self	a method called automatically when any simulation is compiled to provide default values and adjust bounds for any locally created variables which may have unsuitable defaults in their ATOM definitions. Usually the variables selected are those for which the model becomes ill-behaved if given poor initial guesses or bounds (e.g., zero). This method should include statements to run the *default_self* method for each of its locally created (IS_A'd) parts. This method should be written first.
ClearAll	a method to set all the *.fixed* flags for variables in the type to *FALSE*. This puts these flags into a known standard state -- i.e., all are *FALSE*. All models inherit this method from the base model and the need to rewrite it is very, very rare.
specify	a method which assumes all the fixed flags are currently *FALSE* and which then sets a suitable set of *fixed* flags to *TRUE* to make an instance of this type of model well-posed. A well-posed model is one that is square (*n* equations in *n* unknowns) and solvable.
reset	a method which first runs the ClearAll method and then the *specify* method. We include this method because it is very convenient. We only have to run one method to make any simulation well-posed, no matter how its fixed flags are currently set. All models inherit this method from the base model, as with *ClearAll*.

Table 10-1 List of standard methods we insist be added for each of the types in our ASCEND library of type definitions

method	description
values	a method to establish typical values for the variables we have fixed in an application or test model. We may also supply values for some of the variables we will be computing to aid in solving a model instance of this type. These values are ones that we have tested for simulation of this type and found good.
bound_self	a method to update the *.upper_bound* and *.lower_bound* value for each of the variables. ASCEND solvers use these bound values to help solve the model equations. This method should bound locally created variables and then call bound_self for every locally created (IS_A'd) part.
scale_self	a method to update the *.nominal* value for each of the variables. ASCEND solvers will use these nominal values to rescale the variable to have a value of about one in magnitude to help solve the model equations. This method should rescale locally created variables and then call scale_self for every locally created (IS_A'd) part.

be evident from above, not *all* models must have associated methods; our first vessel model did not. It is simply our policy that models in our libraries must have these methods to promote model reuse and to serve as examples of best practices in mathematical modeling.

10.10 METHOD WRITING AUTOMATION

Just hit the button Library.Edit.Suggest methods and tweak the results.

ASCEND will help you write the standard methods. Writing most of the standard methods can be nearly automated once the declarative portion of the model definition is written. Usually, however, some minor tweaking of the automatically generated code is needed. In the Library window, the Edit menu has a "Suggest methods" button. Select a model you have written and read into the library, then hit this button.

In the Display window will appear a good starting point for the standard methods that you have not yet defined. This starting point follows the guidelines in this chapter. It saves you a lot of typing but it is a starting point only. Select and copy the text into the model you are editing, then tailor it to your needs and finish the missing bits. The comments in the generated code can be deleted before or after you copy the text to your model file.

If you have suggestions for general improvements to the generated method code, please mail them to us and include a sample of what the generated code ought to look like *before* the user performs any hand-editing. We aim to create easily understood and easily fixed method suggestions, not perfect suggestions, because procedural code style tastes vary so widely.

CHAPTER 11 THE MODEL LIBRARIES FOR MULTI-COMPONENT, MULTI-PHASE EQUILIBRIUM CALCULATIONS

This chapter describes the models we provide to compute thermodynamic properties for multi-phase, multi-component vapor/liquid mixtures where we assume equilibrium exists among co-existing phases.

11.1 A DESCRIPTION OF THE LIBRARIES

In this section we describe the three libraries, *phases.a4l*, *components.a4l* and *thermodynamics.a4l*. These libraries contain many models, but the end user is only interested in a few of them. Our intention is that these few should be very simple to use, with the complexities buried inside the models.

first the phase definitions

The first contains the models we use to define the phases we allow for a mixture (i.e., vapor, liquid, vapor/liquid, liquid/liquid and vapor/liquid/liquid)[1].

then the components and their data

The second library contains the models having all the component physical properties for the components we include with ASCEND — e.g., there are property values for heat capacity, heat of vaporization, accentric factor and so forth for water, methanol, carbon dioxide, etc. There is also the very extensive list of group contribution data we need to use the UNIFAC method.

and finally the mixture thermodynamic models

The third provides the models we use to compute multi-component mixture thermodynamic properties for phases, such as ideal gas, Pitzer, UNIFAC, and Wilson. The final model in this library is the one to compute equilibrium conditions for multi-component, multi-phase systems. We provide both a constant relative volatility and a rigorous phase equilibrium model, with the ability to switch interactively between which one to use. Thus one can first assume constant relative

1. It should be noted that, while the models will correctly set up the data structures for the liquid/liquid and vapor/liquid/liquid options, we do not really support these alternatives at this time.

volatility to have a better chance to converge and then switch to the version that makes the chemical potential equal for a component in all phases.

11.1.1 THE *PHASES.A4L* LIBRARY

need to create only instances of *phases_data*

The *Phases.a4l* library, see Figure 11-1[2], has only one model in it, **phases_data**. The user creates an instance of this model, specifying which phases are to exist for a stream or holdup and which thermodynamic model the system should use to compute mixture properties for each phase. Compiling this instance then sets up the data structures required to characterize those phases for the system.

For example, suppose we want to model a flowsheet consisting of a single flash unit. Suppose further that we want to allow the feed to the flash unit to be vapor, liquid or vapor/liquid (i.e., 2 phase). The product streams from the flash unit will be a vapor phase mixture and a liquid phase mixture. We would define three instances of the phases_data model, one for each type of phase condition we wish to model. You can find the following statements in the model *testflashmodel* in the library *flash.a4l*.

```
pdV IS_A phases_data('V', 'ideal_vapor_mixture', 'none', 'none');
pdL IS_A phases_data('L', 'none', 'UNIFAC_liquid_mixture', 'none');
pdVL IS_A phases_data('VL', 'ideal_vapor_mixture', 'UNIFAC_liquid_mixture',
                     'none');
```

When compiled, *pdV, pdL* and *pdVL* contain the data structures the thermodynamic models require to model a vapor, liquid and vapor/ liquid stream (or holdup).

2. In this and following figures, we represent each model as a rectangle. On the upper left is the name of the model. In Figure 11-1, the model is phases_data. On the left side we list in order the parameters for the model. These are shared objects a model containing an instance of phases_data will pass to that instance. An example would be

```
pd IS_A phases_data(V, ‘Pitzer_vapor_mixture’, ‘none’, ‘none’)
```

We list the parts defined locally within a model on the right side of the rectangle, including instances of models, atoms and sets. The slanted double-headed arrow indicates a set; thus, phases and other_phases are sets in phases_data.

In Figure 11-3 we show lines connecting a model, call it *A*, to a part within another model, call it *B.part.* The connection is to the sides of both. This type of connection says *B.part* is an instance of model *A*. We also show connections from the bottom of one model, call it *C*, to the top of another, call it *D*; with this connection we indicate that the lower model *D* is a refinement of the upper model *C*.

the phase indicators and types

The first parameter is a character that indicates the phase option desired - 'M', 'V', 'L', 'VL', 'LL' and 'VLL'. 'M' is for a material only stream (no thermodynamic properties are to be computed), 'V' is for vapor and 'L' for liquid. This model always expects the user to supply in the last three parameters an ordered list giving the three single phase mixture models to be used: vapor, liquid1, liquid2. For a non-existent phase, the user should supply 'none' as the model. If there is only one liquid phase, liquid2 will not exist. The allowed models we can use to estimate multi-component phase mixture properties are in the third of the libraries we describe in this chapter, *thermodynamics.a4l*, which we discuss shortly in Section 11.1.3.

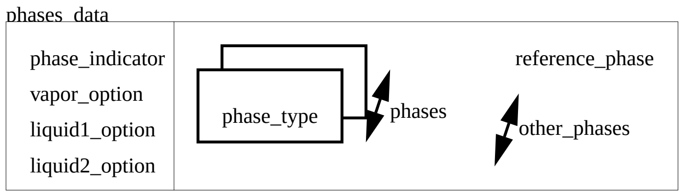

Figure 11-1 *Phases.a4l* models

11.1.2 THE *COMPONENTS.A4L* LIBRARY

In this library (see Figure 11-2) we provide the actual physical property data for the components supplied with ASCEND. The data we provide is that found in the tables at the back of Reid, Prausnitz and Poling, The Properties of Vapors & Liquids, 4th Ed, McGraw-Hill, New York (1986). For a few of the components, we have also identified their UNIFAC groups. We include a few Wilson binary mixture parameters.

need to create only instances of *components_data*

The purpose of this library is similar to the *phases.a4l* library. We wish to provide an easy-to-use model that will set up the data structures for the components in a mixture that the thermodynamic models will use when estimating mixture physical properties. All the user has to do is create an instance of the bottom-most model *components_data*, passing into it a list of the components in the mixture and the name of one of them which is to serve as the reference component. This model, having parts which are instances of the others present in this library, then compiles into the needed data structures.

An example of use is found in the model *testflashmodel* in the library *flash.a4l*:

```
cd IS_A components_data(['n_pentane','n_hexane','n_heptane'],'n_heptane');
```

When compiled *cd* has in it a data structure containing the physical properties for the three species listed.

reference component

The choice of which species to use as the reference component is up to the user. Usually a good choice is one that is plentiful in the mixture, but that need not be so.

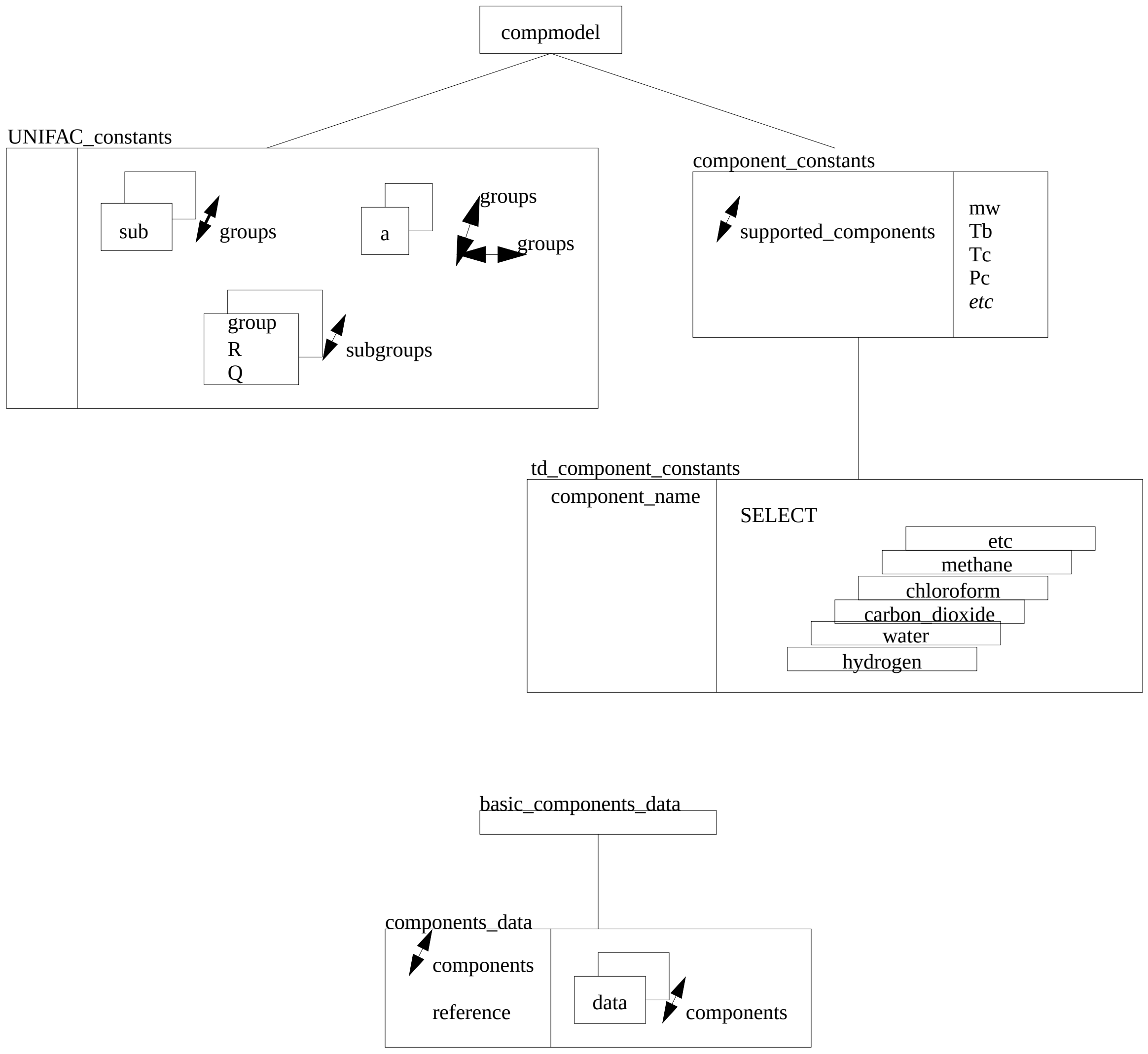

Figure 11-2 *components.a4l* models

adding a new component

One can add more components to this library as follows:

1. add the name of the new component to the list of *supported_components* at the beginning of the model *td_thermodynamic_constants* (part of the WHERE statement that causes the system to output a diagnostic if someone subsequently

misspells the name of a component)

2. add the component data as a CASE to the SELECT statement in *td_thermodynamic_constants* (for an example, look at how it is done for 'methanol')

adding UNIFAC group identifiers

3. Put the UNIFAC group identifiers for the new component into the set subgroups. To illustrate, this statement for methanol is:

```
subgroups      :== ['CH3', 'OH'];
```

You can find all the UNIFAC group identifiers possible in the model *UNIFAC_constants*. Then fill in the vector *nu* with a value for each of these groups (to indicate how many such groups are in the molecule). To illustrate, the values for methanol are:

```
nu['CH3']      :==1;
nu['OH']       :==1;
```

If you are entering the component without identifying its UNIFAC groups, then enter the subgroups statement and define it as empty — i.e., write

```
subgroups      :== [ ];
```

There should then be no entry for *nu* (see the CASE for hydrogen, for example). An activity coefficient estimated by the UNIFAC method will be unity for such a component.

adding Wilson parameters

4. To add Wilson parameters, first fill in the names of the other components for which you are adding data into the set *wilson_set*. For example, this set for methanol might be:

```
wilson_set     :== ['H2O','(CH3)2CO','CH3OH'];
```

Then fill in lambda and energy parameters into the arrays *lambda* and *del_ip*, one for each of the other components. Again, to illustrate, these arrays for methanol would be:

```
lambda['H2O']     :==0.43045;
lambda['(CH3)2CO']:==0.77204;
lambda['CH3OH']   :==1.0;
del_ip['(CH3)2CO']:==2.6493E+002 {J/g_mole};
del_ip['H2O']     :==1.1944E+002 {J/g_mole};
del_ip['CH3OH']   :==0.0 {J/g_mole};
```

Finally for each of these other components, go to its CASE statement, add the name of the new component to its *wilson_set* and then add statements to set the corresponding lambda and energy data. BEN, IS THIS RIGHT????

If you are not adding any Wilson data, enter the statement:

```
wilson_set     :== [ ];
```

11.1.3 THE *THERMODYNAMICS.A4L* LIBRARY

create instances only of *phase_partials* and *thermodynamics*

Figure 11-3 shows all the models in this library and how they are related to each other. There are two models in this library that the user has to worry about: *phase_partials* and *thermodynamics*. The user creates one instance of *thermodynamics* for every stream or holdup in a process model. Each instance, when compiled has parts which are instances of the other models in this library and which are create the equations to compute the thermodynamic properties for a multi-component, multi-phase mixture.

However, the user must pass each instance of a thermodynamics model an array of instances of *phase_partials*, one for each phase in the mixture. One *phase_partials* model must exist for each phase in each stream or holdup in the process model as it provides the equations modeling that phase.

Each of the models in the array of phase_partials must be refined to be one of the possible models for computing properties for a single phase mixture, i.e., one of the models lying below the *phase_paritals* model in Figure 11-3: *ideal_vapor_mixture*, *Pitzer_vapor_mixture*, *UNIFAC_liquid_mixture* or *Wilson_liquid_mixture*. I

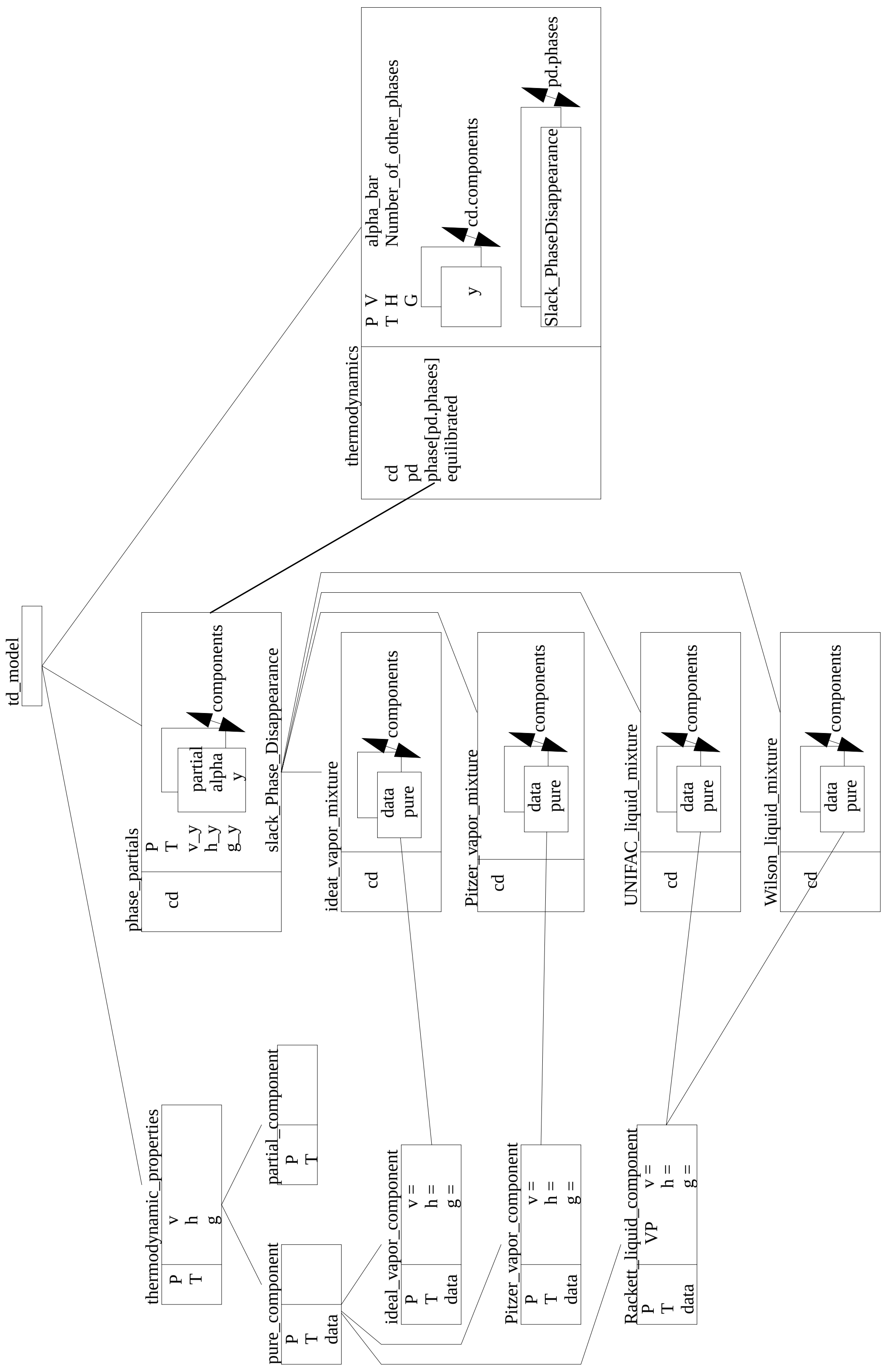

Figure 11-3 Models in *thermodynamic.a4l*

11.1.3.1 CREATING AN INSTANCE OF A *PHASE_PARTIALS* ARRAY

The information in an instance of a *phases_data* model allows us to construct this array of *phase_partials*. We extract the following code from the library *stream_holdup.a4l* to illustrate how we have created such a model, given a phases_data model.

```
MODEL select_mixture_type(
    cd WILL_BE components_data;
    type WILL_BE symbol_constant;
) REFINES sh_base;
   phase IS_A phase_partials(cd);
   SELECT (type)
      CASE 'ideal_vapor_mixture':
         phase IS_REFINED_TO ideal_vapor_mixture(cd);
      CASE 'Pitzer_vapor_mixture':
         phase IS_REFINED_TO Pitzer_vapor_mixture(cd);
      CASE 'UNIFAC_liquid_mixture':
         phase IS_REFINED_TO UNIFAC_liquid_mixture(cd);
      CASE 'Wilson_liquid_mixture':
         phase IS_REFINED_TO Wilson_liquid_mixture(cd);
      OTHERWISE:
   END SELECT;
   boundwidth IS_A bound_width;
   ...
   ...
   ...

END select_mixture_type;

MODEL stream( .......
   ...
   ...
   ...

   FOR j IN phases CREATE
      smt[j] IS_A select_mixture_type(cd, pd.phase_type[j]);
   END FOR;
   FOR j IN phases CREATE
      phase[j] ALIASES smt[j].phase;
   END FOR;
   state IS_A thermodynamics(cd, pd, phase, equilibrated);
   ...
   ...
   ...
   ...
```

cannot directly embed *SELECT* statements in *FOR* loops

We had to be a bit tricky, but we hope we have not been so devious that you cannot understand what we have done if we explain it to you here. Look first at the code we extracted from the model *stream*. The models *cd* and *pd* are instances of a *components_data* and a *phases_data* model respectively. If we look inside *pd*, we will find it contains an array called *phase_type*, with one entry for each phase that gives the type (name) of the model to be used to set up the equations for that phase. ASCEND does not allow *SELECT* statements to be embedded directly within a *FOR* loop — thus we need a bit of deviousness. For each phase *j* we create *smt[j]* as an instance of a *select_mixture_type* model. We parameterize the *select_mixture_type* with the components data *cd* and the type (name) *pd.phase_type[j]* of the model to be used to generate its equations. Then we embed the select statement within the *select_mixture_type* model, something ASCEND does allow.

The model *select_mixture_type* appears first in this code. It uses the *type* (name) it is passed to select and then to instance the desired refinement of the *phase_partials* model.

Returning to the code extracted from the *flash* model, the second *FOR* loop creates the desired array by aliasing the array element *phase[j]* with the phase model created within the corresponding *smt* instance.

disappearing phases

The multi-phase model handles the case where a phase disappears by using a complementarity formulation. This formulation relaxes the constraint for a phase that its mole fractions must sum to unity when it disappears. Thus the vapor/liquid model will correctly alter the model to handle the situation when the mixture becomes a superheated vapor or a subcooled liquid.

11.1.3.2 CREATING AN INSTANCE OF A *THERMODYNAMICS* MODEL

We are now ready to create an instance of a *thermodynamics* model. When compiled this instance contains all the equations needed to estimate the phase conditions for a multi-phase, multi-component mixture assuming equilibrium exists among the phases. The following line of code, extracted from the *stream* model referred to above, illustrates its use:

```
state IS_A thermodynamics(cd, pd, phase, equilibrated);
```

where *cd* is an instance of a *components_data* model, *pd* of a *phases_data* model, *phase* an array of instances of *phase_partials*, and *equilibrated* a *boolean* variable. When *equilibrated* is *FALSE*, the model will generate the equations assuming constant relative volatilities (the user must estimate these volatilities). When *TRUE*, the model generates the equations assuming the chemical potentials for a component are equal in all phases.

11.2 USING THE THERMODYNAMICS MODELS

There are several libraries of models which use the libraries we have just described. The first library to examine is stream_holdup.a4l. This library contains steady-state models for a stream and a holdup. The following gives the parameter list for a user to create an instance of a stream.

11.2.1 STREAMS AND HOLDUPS

```
MODEL stream(
   cd WILL_BE components_data;
   pd WILL_BE phases_data;
   equilibrated WILL_BE boolean;
) REFINES sh_base;
```

The model sh_base is a dummy model to tie all models into this library back to a common root model. The user need do nothing because of this refinement. What you should note is that all you need to do to create a stream is create a *components_data* model and a *phases_data* model. One supplies the boolean variable *equilibrated* as a variable that one can set interactively or in a method or a script when running the model to decide how to model equilibrium, as we have discussed above. A holdup is equally as easy to model.

11.2.2 FLASH UNITS AND VARIANTS THEREOF

From streams and holdups, we can move on to unit operation models. The library flash.a4l provide us with a flash model. The parameter list for the flash model is:

```
MODEL vapor_liquid_flash(
   Qin WILL_BE energy_rate;
   equilibrated WILL_BE boolean;
   feed WILL_BE stream;
   vapout WILL_BE stream;
   liqout WILL_BE stream;
) WHERE (
   feed, vapout, liqout WILL_NOT_BE_THE_SAME;
   feed.cd, vapout.cd, liqout.cd WILL_BE_THE_SAME;
   vapout.pd.phase_indicator == 'V';
   liqout.pd.phase_indicator == 'L';
   (feed.pd.phase_indicator IN ['V','L','VL','VLL']) ==
TRUE;
) REFINES flash_base;
```

Again we see that to create a *flash* unit, we need to create the variable *Qin* for the heat input to the unit, a boolean *equilibrated* and three streams, *feed*, *vapout* and *liqout*. The three streams must all be different streams. They must have the same components in them. The stream *vapout* must be a vapor stream and the stream *liqout* a liquid stream. The feed stream can be of any kind.

Hopefully with the above information, creating a flash unit should not now seem particularly difficult.

If you examine this library further, you will see it contains models which are variations of the flash unit for: *detailed_tray*, *tray*, *feed_tray*, *total_condenser* and *simple_reboiler*.

11.2.3 DISTILLATION COLUMNS

We provide two libraries that allow you to model distillation columns: *column.a4l* and *collocation.a4l*. The library *column.a4l* first models a tray stack and then a simple column using that model. A third model extracts the profiles for pressure, temperature, a parameter that indicates the deviation from constant molar overflow conditions, total vapor and liquid flows and component compositions against tray number. This information may then be used for plotting these profiles using the ASCEND plotting capability.

The library *collocation.a4l* provides collocation models for simple columns. With collocation models, one models composition profiles as smooth functions of tray number in a column section. Columns with a large number of trays are modeled with relatively small collocation models. Also the number of trays becomes a continuous variable, aiding in optimization studies where the number of trays in each section is to be computed.

11.2.4 DYNAMIC UNIT MODELS

ASCEND contains models for simulating the dynamic behavior of units. Their use is described in Chapter xxxx.

11.3 DISCUSSION

We have presented a description of the libraries that allow one to model the equations providing thermodynamic properties for multi-component, multi-phase mixtures when one assume equilibrium exists among co-existing phases. With this description, we hope that these models become much less difficult to use. We end this chapter by describing other libraries that build on the property estimation libraries, models for streams and holdups, for flash units and variations thereof, and for columns.

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.

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

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

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.

CHAPTER 13 CREATING CONDITIONAL MODELS IN ASCEND

In this chapter, we describe how one can create conditional models in the ASCEND environment.

what is a conditional model ?

Formally, we consider as a conditional model any problem in which the domain of validity of alternatives sets of equations depends on one or more discrete conditions; conditions can be expressed in terms of any logical, integer, or binary variables or constants. For instance, think of a case in which you need to solve a system of equations including some sort of numerical correlation (correlation data for physical properties, for instance). You realize that the coefficients of your correlation change with the value of some other variables of the problem (temperature, pressure, etc.). You have a conditional model.

ASCEND support three modeling capabilities for the efficient development of conditional models:

- Conditional configuration of the model structure.
- Conditional compilation.
- Conditional execution of the procedural code of methods.

In the following sections we describe the modeling tools for the performance of each of these tasks: the WHEN statements for the conditional configuration of a model structure, the SELECT statement for conditional compilation, and the SWITCH statement for conditional execution of procedural statements.

13.1 THE WHEN STATEMENT: CONDITIONAL CONFIGURATION OF THE MODEL STRUCTURE

We start by defining the syntax for the WHEN statement:

```
eq1_identifier: definition_of_equation_1;
model1_identifier: definition_of_model_1;
```

```
    ⋮
modeln_identifier: definition_of_model_n;

WHEN (list_of_variables)
   CASE list_of_values_1:
            USE eq1_identifier;
   CASE list_of_values_2:
            USE model1_identifier;
    ⋮
   OTHERWISE:
            USE modeln_identifier;
END WHEN;
```

observations about the WHEN statement

The following are important observations about the WHEN statement:

1 A list of variables is used to define the applicability of each of the alternative configuration. The variables in this list can be of any type among boolean, integer or symbol or any combination of them. Note that the list is surrounded by rounded parentheses: (). We do that to emphasize that order matter in such a list — consistent with the use of rounded parentheses throughout ASCEND.
2 The values in this list for each of the cases are in one to one correspondence with the variables in the list.
3 Names of arrays of models or equations are allowed inside the scope of each CASE.
4 All the objects and equations used in the different CASEs of a WHEN statement are compiled. However, the objects (the variables and relations defined in it) of a particular CASE will only become part of our mathematical problem if the values in the list of values of that CASE match the current values of the variables in the list of variables. Practically speaking, to "USE" an object (model) means that the variables and equations contained in that object will become an active part of the system of nonlinear equations representing the current configuration of the problem.

There are two different ways in which the WHEN statement can be used.:

select among alternatives

- First, the WHEN statement can be used to select a configuration of a problem among several alternative configurations. This chapter is mainly concerned with this type of simpler and more common application.

conditional program

- Second, in combination with logical relations, the WHEN statement can be used for conditional programming — that is, a problem in which the system of equations to be solved depends on the solution of the problem.

13.1.1 THE SIMPLEST EXAMPLE

Assume that you want to solve a system of equations in which two correlations are possible for the calculation of a variable. Of course, you could create two simple models, each of them including one of the alternative equations. You could also use the WHEN statement to create only one model, in which you could include both alternatives. In this latter case you will be able to switch readily from one alternative to the other without recompiling. Look at the following simple case:

```
laminar  IS_A boolean;
Re,f     IS_A factor;

invariant: sqrt(f) * Re = 0.00034576;
low_flow: Re = 64/f;
high_flow: Re = (0.206307/f)^4;

WHEN (laminar)
   CASE TRUE:
      USE low_flow;
   CASE FALSE:
      USE high_flow;
END WHEN;
```

The model contains three equations, all of which are compiled. There is one equation (named `invariant`) which is not used in any of the CASEs of the WHEN statement. Such an equation is always part of the mathematical problem that we are trying to represent. On the other hand, the equations `low_flow` and `high_flow` are conditional equations because they are used in a CASE of the WHEN statement. The equations `low_flow` and `high_flow` are part of the mathematical problem only when the value of the boolean variable `laminar` matches the value of the list of values of the CASE in which they are defined. If we decide that we need to use the equation `low_flow`, then we have to give the value of TRUE to the boolean variable `laminar`. if we decide to use the equation `high_flow`, then we have to give the value of FALSE to the boolean variable `laminar`. Note that the value of the variable `laminar` can be modified as many times as the user wishes. In this way, the user may readily switch from one configuration to the other. In either of the CASEs, the resulting system of equations contains two equations (`invariant` and either `low_flow` or `high_flow`) in two variables (Re and f).

We could have used another kind of variable in the list of variables with exactly the same result. In the following example, an integer is used instead of a boolean. With an integer variable, we can have as many distinct CASEs as we wish inside a WHEN statement.

```
laminar  IS_A integer;
Re,f     IS_A factor;

invariant: sqrt(f) * Re = 0.00034576;
low_flow: Re = 64/f;
high_flow: Re = (0.206307/f)^4;

WHEN (laminar)
   CASE 1:
      USE low_flow;
   CASE 2:
      USE high_flow;
END WHEN;
```

13.1.2 A SECOND EXAMPLE

```
method   IS_A symbol;
simplified_flash  IS_A VLE_flash;
rigorous_flash    IS_A td_VLE_flash;

WHEN (method)
   CASE ‘rigorous’:
      USE rigorous_flash;
   CASE ‘simplified’:
      USE simplified_flash;
END WHEN;
```

For this example, we have exactly the same capability as in our previous simplest example; however, here the objects named inside the WHEN statement are models and not relations. Also, the decision is based in the value of a symbol variable: `method`. As mentioned before, practically speaking, to “USE” an object (model) means that the variables and equations contained in that object will become an active part of the system of nonlinear equations representing the current configuration of the problem.

13.2 THE SELECT STATEMENT: CONDITIONAL

COMPILATION

Aside from the flexibility that conditional statements (such as the WHEN statement) gives to the configuration of a model structure, another application of conditional tools is the economy of programming. An example commonly occurring in engineering is the selection of the thermodynamic model to be used for equilibrium calculations. In general, it is convenient to code all of the alternative methods so that, depending on the species appearing in the equilibrium system, we can select the most appropriate method.

In this kind of problem, the decision as to which configuration we are going to use can be made before we compile the model. We would like to compile only the configuration appropriate for the problem rather than compiling all available configurations.

The SELECT statement incorporates conditional compilation into the ASCEND system. While this conditional tool is flexible enough to represent all of the alternatives, its presence will indicate that only those alternatives consistent with the model data will be available after compilation.

Even though the syntax for the SELECT statement is similar to that described for the WHEN statement, we nned to highlight some important differences:

- In the WHEN statement the declaration of the object is external to the conditional statement since of all the alternatives are going to be created anyway. In the SELECT statement, the actual declaration of an object (or any other declarative statement affecting objects) is done within each CASE of the conditional statement, explicitly discriminating among the alternative statements. Thus parts of a particular kind can exist in only one case within a select statement.
- The selection among alternatives in the SELECT statement depends on constant boolean variables, constant integer variables or constant symbols. Since these values imply a one time structural decision, they must not be modified during the solution of the problem. That is why they have to be constants.

The following is the syntax used for the conditional compilation tool:

```
defintion_of_constants;
assignment_of_constant_values;
```

```
   SELECT (list_of_constants)
      CASE list_of_values_1:
               list1_of_declarative_statements;
      CASE list_of_values_2:
               list2_of_declarative_statements;
       ⋮
      OTHERWISE:
               listn_of_declarative_statements;
   END SELECT;
```

Summarizing, the SELECT statement provides the capability of conditional compilation. It allows the representation of structural alternatives pursuing economy in programming, but, since only the desired data structure is created, it does not affect the computational requirements of the model.

13.2.1 A SIMPLE EXAMPLE

The following example shows an ASCEND model which is similar to that shown in the previous section. The difference is that we use the SELECT statement rather than the WHEN statement. This time, the symbol `method` is a constant, and, once it is defined, its value will not change. That value will always be a user decision. Also, note that the definition of the objects is done inside the SELECT statement. For this example, since the value of the symbol `method` is 'rigorous', the system will compile only the list of statements in the first CASE.

```
method   IS_A symbol_constant;
method :== 'rigorous';

SELECT (method)
   CASE 'rigorous':
      rigorous_flash IS_A td_VLE_flash;
   CASE 'simplified':
      simplified_flashIS_A VLE_flash;
END SELECT;
```

13.3 THE SWITCH STATEMENT: CONDITIONAL EXECUTION OF PROCEDURAL CODE

Because of the use of conditional statements in the declarative description of a model, a similar feature must also exist to give the user the ability to program the conditional execution of methods. For instance, each alternative configuration of a model may require different initialization and a different selection of the independent variables for the solution process. Hence, ASCEND has the following conditional procedural SWITCH statement:

```
   SWITCH (list_of_variables)
      CASE list_of_values_1:
               list1_of_procedural_statements;
      CASE list_of_values_2:
               list2_of_procedural_statements;
       ⋮
      OTHERWISE:
               listn_of_procedural_statements;
   END SWITCH;
```

This statement has the same meaning as conditional statements that exist in procedural modeling languages such as C and FORTRAN. The procedural statements in each of these cases do not involve new object definitions, they are only useful for the numerical processing of objects already created.

13.3.1 A SIMPLE EXAMPLE

The use of the SWITCH statement for the conditional execution of procedural code is illustrated below. In this example, the value of the variable `ave_alpha` is set to 1.5 only if the value of the symbol `method` is ‘simplified’. If the value of the symbol `method` is ‘rigorous’, then a procedure called `adiabatic` is executed instead.

```
METHODS
   METHOD values;
      RUN reset;
      SWITCH (method)
         CASE ‘rigorous’:
            RUN adiabatic;
         CASE ‘simplified’:
            ave_alpha := 1.5;
      END SWITCH;
   END values;
```

CHAPTER 14 BOUNDARY VALUE PROBLEMS

The subject of formulating and solving boundary value problems (BVP) is very large. In this chapter we demonstrate how to model and solve the boundary value formulation of any number of differential equations in one independent variable and possibly subject to algebraic constraints. In physical systems, the independent variable is typically time or distance. This chapter does not cover problems involving two or more independent variables.

We begin with a very simple ode and compare a number of numerical methods. We then investigate a nonlinear model of water in a tank. We shall see in the end that a single model of the physical system can be used for the initial value problem (IVP) as well as the BVP. This is a very important result because hard, algebraically constrained BVPs from engineering problems often cannot be solved without initialization from an approximate solution obtained with an IVP method.

14.1 BVP.A4L

A basic example is found in ascend4/models/plotbvp.a4s which shows how to define, solve, and plot a model. We will explain it in some detail here in the near future.

.

Index