March 30, 1994

The ASCEND modeling language

Molar stream example

We shall start our description of the ASCEND modeling language by presenting some examples. Let us start with the model of what superficially seems a simple concept: a stream characterized only by the flows for the components in it. We will build on this model to construct that for some simple unit operations and then for a total flowsheet.

Molar stream

To model a simple molar stream, we must decide on the concepts on which it is to be constructed. A molar stream is one for which one lists only the molars flowrate and the mole fraction for each of the components in it as well as a total flowrate.

We shall apply some hard learned knowledge about modeling streams and insist that the intensive properties for a stream be collected together into a single concept we shall call its state. Then a stream will be its state plus its extensive properties. Intensive variables are those that are not quantity dependent. Here the only intensive variables are those describing the composition of the stream, i.e., the mole fractions for each of the components. The molar flowrates and the total molar flowrate are extensive variables whose values would double if we were to double the size of the stream.

We will call this collection of intensive variables for a molar stream a *molar mixture*. This model is in Fig. 1. In it we define a set of mole fractions, one for each member of a set of component names.

Model structure

Declarative

This model has a number of statements in it that we need to discuss. It is headed by a line stating we are defining a new model for ASCEND. The next two lines define the parts in the model and what type of thing each is. One item is a set of components names, each of which is of type *string*. A string is a built-in type in ASCEND; it is an unbroken sequence of up to 36 alphanumeric characters enclosed within two single quotes. For each member `i` in this set of components, we define a variable `y[i]` whose interpretation is to be the mole fraction for that species in the stream. It is of type *fraction.* The definition for fraction is in the library file called atoms.lib which is available as a separate document. Atoms.lib contains all definitions for a large number of different variable types and should be referenced before starting the writing of a model.

Fig. 2 shows a typical variable type definition, one for molar_rate. One can define a new type of variable at any time by including a new *atom* definition. An atom definition may be thought of as the most elementary model definition in ASCEND and is used solely to define variable types. All other type definitions in ASCEND are called *models*.

The definition for the type *molar_rate* illustrates the idea of inheritance, a key capability of the ASCEND language. We see that molar_rate is a refinement of a previously defined type *generic_real*. Basically one can define a model or an atom to inherit all the

statements of a previously defined model or atom and then add further statements to complete the new type definition. Thus the new item is everything the older one is plus more.

```
MODEL mixture;

   components                              IS_A set OF string;
   y[components]                           IS_A fraction;

   SUM(y[i] | i IN components) = 1;


   INITIALIZE

      PROCEDURE clear;
         y[components].fixed               := false;
      END clear;

      PROCEDURE specify;
         y[components].fixed               := true;
         y[CHOICE[components]].fixed       := false;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
      END scale;

END mixture;
```

Figure 1. An ASCEND model for the intensive variables in a molar stream

```
ATOM molar_rate REFINES generic_real
    DIMENSION mole/t
    DEFAULT 100.0{lb_mole/hour};
    low_bound := 0.0{lb_mole/hour};
    nominal := 100.0{lb_mole/hour};
END molar_rate;
```

Figure 2. Definition of the variable type *molar_rate*

Returning to our mixture model, the mole fractions must add to unity. We add this constraint using the SUM function which is built into ASCEND. The sum function adds together all of its arguments; there is no limit on the number of arguments it may have.

Procedural

So far everything is simple. We now see a section of the model entitled INITIALIZE, and, in this section, we find several PROCEDURE definitions. Any set of procedures may be included. A modeler can ask that the ASCEND system to execute these procedures at any time and in any order while debugging and solving. They are executed ONLY on request. They are true procedures; the statements execute in order and the value of a variable on the left-hand-side of a statement is replaced by the value of the expression on the right.

The particular procedures included here are based on experience and are becoming a standard for all models we write. Let us make this standard into a modeling rule as follows.

> Good modeling practice in ASCEND: Attach procedures *clear, specify, reset* and *scale* to all models created in ASCEND as a part of good modeling practice.

Doing so will force a modeler to consider these issues carefully. Also, from our experience in creating and debugging models, these issues, once resolved, are an excellent starting place for all other uses of a model.

Associated with every computable variable in ASCEND is a boolean flag indicating if its value is to be fixed or not when solving. If *true*, the variable value is held at its current value while solving; if *false* its value is computed using the equations defining the model. In the *clear* procedure we set the flags for all computable variables to *false*.

In the *specify* procedure, we set exactly enough of the flags to true to make the model square (i.e., *n* equations in *n* unknowns). This model has exactly one equation. The fixed flag for each of the compositions is set to true except for one which can be computed. The CHOICE function selects exactly one of the elements in the set mentioned, the choice being arbitrary.

The *reset* procedure simple runs the clear procedure and then the specify procedure to return the model fixed flags to a known standard state.

Finally, we use the *scale* procedure to allow extensive variables to be scaled according to their current values. This model only contains intensive variables so the *scale* procedure is empty. As a part of good modeling practice, we do insist that the procedure exists even if it is empty.

Molar stream example

We are now ready to create the molar stream model, Fig. 3.

The molar stream model contains a part called state whose type is a molar mixture, a type we just defined. It also has variables to represent the total molar flowrate for the stream and the molar flows for each of the stream components.

We see our first ARE_THE_SAME statement. The set of component names defined for the stream and the set of component names inside the state are to be exactly the same set.

In the current version of ASCEND, these sets are stored in the exact same storage locations. The one set has two names in the molar stream model: *components* and *state.components*.

```
MODEL molar_stream;

   components                                IS_A set OF string;
   state                                     IS_A mixture;
   Ftot, f[components]                       IS_A molar_rate;
   FOR i IN components CREATE
      f[i] = state.y[i]*Ftot;
   END;

   components, state.components              ARE_THE_SAME;

   INITIALIZE

      PROCEDURE clear;
         RUN state.clear;
         Ftot.fixed                          := false;
         f[components]                       := false;
      END clear;

      PROCEDURE specify;
         f[components].fixed                 := true;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN state.scale;
         Ftot.nominal := 0.0001 {kg_mole/s} + abs(Ftot);
         f[components].nominal:= Ftot.nominal;
      END scale;

END molar_stream;
```

Figure 3. An ASCEND model for a molar stream

Qualified names

ASCEND allows access into parts within a model through the use of qualified names. We placed a design requirement on the language that the users of a model definition would define the interface to that model; i.e., users, and not the model creator, would be able to select which variables within a model they wish to access. In this manner the same model definition can satisfy different uses without itself being touched.

The approach is that one can attach to any instance within a part by constructing a qualified name for it. The name has the structure:

part_name.inner_instance_name

Such a naming scheme can be repeated to any depth, allowing us to go, so to speak, to the bottom-most layers of a model and use an instance appearing there at the outer-most level.

We used qualified names in the model *molar_stream* above. This model contains a part called *state* which is an instance of the model *mixture*. Mixture has a part called *components*, the set of names for the components in it. Molar_stream also has a set called *components*, and we wanted these two sets to be the same set. We needed a means to name the set called components within the part called state which we did by constructing the name

state.components

and using it in an ARE_THE_SAME statement.

In the procedures, we accessed the fixed attribute for variables when we wanted to select which variables are to be fixed during a computation. An example was

f[components].fixed := true;

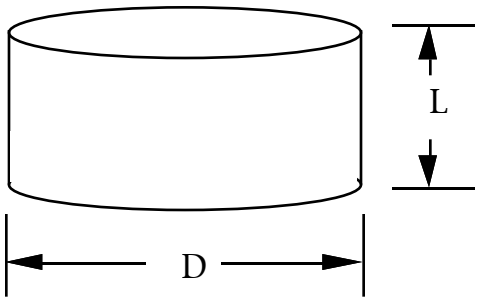
which is interpreted as fixing the molar flowrates for the whole set of components.

Constant relative volatility flash example

Fig. 4 is a simple relative volatility flash model. It uses the previously defined concept of a stream in its definition. It could be interesting to compare the equations it will generate to the ten we listed for the simple two component model earlier.

Homework 8

1. Create an ASCEND model to compute the mass of metal in a thin walled cylindrical tank shown the following figure. The tank has flat ends. Wall thickness is t.



2. Write an ASCEND model for the two-species flash model we used earlier to describe modeling. Include all parts for it.

```
MODEL flash;

   feed, vap, liq                          IS_A  molar_stream;

   feed.components,
   vap.components,
   liq.components                          ARE_THE_SAME;

   alpha[feed.components],
   ave_alpha                               IS_A  factor;

   vap_to_feed_ratio                       IS_A  fraction;

   vap_to_feed_ratio*feed.Ftot = vap.Ftot;

   FOR i IN feed.components CREATE
      cmb[i]:  feed.f[i] = vap.f[i] + liq.f[i];
      eq[i]:   vap.state.y[i]*ave_alpha = alpha[i]*liq.state.y[i];
   END;

  INITIALIZE

   PROCEDURE clear;
      RUN feed.clear;
      RUN vap.clear;
      RUN liq.clear;
      alpha[feed.components].fixed         :=    false;
      ave_alpha.fixed                      :=    false;
      vap_to_feed_ratio.fixed              :=    false;
   END clear;

   PROCEDURE seqmod;
      alpha[feed.components].fixed         :=    true;
      vap_to_feed_ratio.fixed              :=    true;
   END seqmod;

   PROCEDURE specify;
      RUN seqmod;
      RUN feed.specify;
   PROCEDURE specify;

   PROCEDURE reset;
      RUN clear;
      RUN specify;
   END reset;

   PROCEDURE scale;
      RUN feed.scale;
      RUN vap.scale;
      RUN liq.scale;
   END scale;

END flash;
```

Figure 4. Simple flash model written in the ASCEND language

Complete simple flowsheet example

Fig. 5a-h gives the code (plus the models *mixture*, *molar_stream* and *flash* above) for an entire simple flowsheet. We add models for a *mixer*, a *reactor* and a *splitter*. The model *flowsheet* configures the entire flowsheet. Test routines complete the model. We also add a computational controller to allow us the specify the degree of conversion for the reaction in the reactor in terms of component B.

Note that comments in ASCEND start with *(** and end with ***)*. You should be able to read the code for this model and understand it without further explanation.

Merging to configure complex models

Another important concept in modeling is the ability to configure a complex model by putting a number of simpler parts together to form the model. The model called *flowsheet* in Fig. 5e illustrates the power of this form of configuration. It declares first that there are to be four parts to the flowsheet: a mixer called m1, a reactor called r1, a flash called fl1 and a splitter called sp1. The mixer is to have two inputs while the splitter has two outputs. We configure the flowsheet by *merging* streams using the ASCEND operator called ARE_THE_SAME. The output stream from the mixer is exactly the same stream as the feed to the reactor. Making them the same is done in ASCEND during compilation by assigning the same storage locations for the same variables in both streams. Also, ASCEND allows only one of the two models to create its equations to avoid the production of redundant equations.

```
MODEL mixer;

   n_inputs    IS_A                        integer;
   feed[1..n_inputs], out                  IS_A  molar_stream;

   feed[1..n_inputs].components,
   out.components                          ARE_THE_SAME;

   FOR i IN out.components CREATE
      cmb[i]: out.f[i] = sum(feed[1..n_inputs].f[i]);
   END;

   INITIALIZE

      PROCEDURE clear;
         RUN feed[1..n_inputs].clear;
         RUN out.clear;
      END clear;

      PROCEDURE seqmod;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed[1..n_inputs].specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed[1..n_inputs].scale;
         RUN out.scale;
      END scale;

END mixer;
```

Figure 5a. Mixer model for flowsheet example

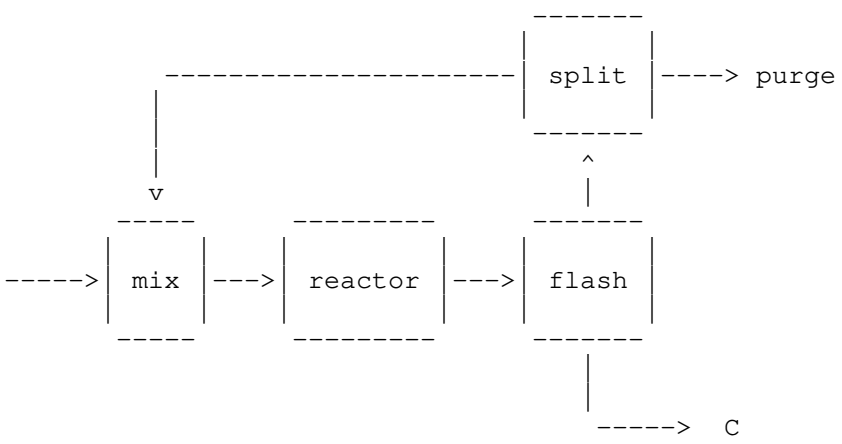
```
MODEL reactor;

   feed, out                              IS_A  molar_stream;
   feed.components, out.components        ARE_THE_SAME;

   turnover                               IS_A  molar_rate;
   stoich_coef[feed.components]           IS_A  factor;

   FOR i IN feed.components CREATE
      out.f[i] = feed.f[i] + stoich_coef[i]*turnover;
   END;

  INITIALIZE

      PROCEDURE clear;
         RUN feed.clear;
         RUN out.clear;
         turnover.fixed                         :=    false;
         stoich_coef[feed.components].fixed     :=    false;
      END clear;

      PROCEDURE seqmod;
         turnover.fixed                         :=    true;
         stoich_coef[feed.components].fixed     :=    true;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed.scale;
         RUN out.scale;
         turnover.nominal := turnover.nominal+0.0001 {kg_mole/s};
      END scale;

END reactor;
```

Figure 5b. Reactor model for simple flowsheet example

```
MODEL splitter;

   n_outputs                                IS_A  integer;
   feed, out[1..n_outputs]                  IS_A  molar_stream;
   split[1..n_outputs]                      IS_A  fraction;

   feed.components,
   out[1..n_outputs].components             ARE_THE_SAME;

   feed.state,
   out[1..n_outputs].state                  ARE_THE_SAME;

   FOR j IN [1..n_outputs] CREATE
      out[j].Ftot = split[j]*feed.Ftot;
   END;

   sum(split[1..n_outputs]) = 1.0;

   INITIALIZE

      PROCEDURE clear;
         RUN feed.clear;
         RUN out[1..n_outputs].clear;
         split[1..n_outputs-1].fixed      :=    false;
      END clear;

      PROCEDURE seqmod;
         split[1..n_outputs-1].fixed      :=    true;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed.scale;
         RUN out[1..n_outputs].scale;
      END scale;

END splitter;
```

Figure 5c. Splitter model for simple flowsheet example

```
(*

The following example illustrates equation based modeling using the
ASCEND system.  The process is a simple recycle process.
```




```
*)
```

Figure 5d. Diagram of simple flowsheet for simple flowsheet example

```
MODEL flowsheet;

   m1                                    IS_A  mixer;
   r1                                    IS_A  reactor;
   fl1                                   IS_A  flash;
   sp1                                   IS_A  splitter;

(* define sets *)

   m1.n_inputs                           :=    2;
   sp1.n_outputs                         :=    2;

(* wire up flowsheet *)

   m1.out, r1.feed                       ARE_THE_SAME;
   r1.out, fl1.feed                      ARE_THE_SAME;
   fl1.vap, sp1.feed                     ARE_THE_SAME;
   sp1.out[2], m1.feed[2]                ARE_THE_SAME;

   INITIALIZE

      PROCEDURE clear;
         RUN m1.clear;
         RUN r1.clear;
         RUN fl1.clear;
         RUN sp1.clear;
      END clear;

      PROCEDURE seqmod;
         RUN m1.seqmod;
         RUN r1.seqmod;
         RUN fl1.seqmod;
         RUN sp1.seqmod;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN m1.feed[1].specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN m1.scale;
         RUN r1.scale;
         RUN fl1.scale;
         RUN sp1.scale;
      END scale;

END flowsheet;
```

Figure 5e. Code for simple flowsheet

```
MODEL controller;

   fs                                    IS_A  flowsheet;
   conv                                  IS_A        fraction;
   key_components                        IS_A        string;

   fs.r1.out.f[key_components] = (1-conv)*fs.r1.feed.f[key_components];

   INITIALIZE

      PROCEDURE clear;
         RUN fs.clear;
         conv.fixed                      :=    false;
      END clear;

      PROCEDURE specify;
         RUN fs.specify;
         fs.r1.turnover.fixed            :=    false;
         conv.fixed                            :=    true;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
         END reset;

      PROCEDURE scale;
         RUN fs.scale;
      END scale;

END controller;
```

Figure 5f. Code for controller

```
MODEL test_flowsheet REFINES flowsheet;

   m1.out.components                        :=    ['A','B','C'];

   INITIALIZE

      PROCEDURE values;
         m1.feed[1].f['A']                  :=    0.5  {kg_mole/s};
         m1.feed[1].f['B']                  :=    0.95 {kg_mole/s};
         m1.feed[1].f['C']                  :=    0.0  {kg_mole/s};

         r1.stoich_coef['A']                :=    0;
         r1.stoich_coef['B']                :=    -1;
         r1.stoich_coef['C']                :=    1;

         fl1.alpha['A']                     :=    12.0;
         fl1.alpha['B']                     :=    10.0;
         fl1.alpha['C']                     :=    1.0;
         fl1.vap_to_feed_ratio              :=    0.9;
         fl1.ave_alpha                      :=    5.0;

         sp1.split[1]                       :=    0.01;

         fl1.liq.Ftot                       :=    m1.feed[1].f['B'];
      END values;

END test_flowsheet;
```

Figure 5g. Test code for simple flowsheet

```
MODEL test_controller REFINES controller;

   fs       IS_REFINED_TO                  test_flowsheet;
   key_components                          :=    'B';

   INITIALIZE

      PROCEDURE values;
         RUN fs.values;
         conv                             :=    0.07;
      END values;

END test_controller;
```

Figure 5h. Code to test controller

Inheritance and deferred binding in modeling

The flowsheet example illustrates inheritance (using the REFINES operator) in ASCEND. It does not illustrate deferred binding, another of the modeling concepts we feel is necessary in equation-based modeling. Model and atom definitions in ASCEND can establish a refinement hierarchy of definitions. For example, Fig. 6 gives the code for five related models. There is a model hierarchy implied in these statements which we can express either using the indented format of Fig. 6 or in graphical form as shown in Fig. 7. The indenting in Fig. 6 is strictly a matter of style in displaying the code.

```
MODEL a;
  statements defining model a
END a;

   MODEL b REFINES a;
     statements defining model b
   END b;

      MODEL c REFINES a;
        statements defining model c
      END d;

   MODEL d REFINES a;
     statements defining model d
   END d;

MODEL e;
  statements defining model e
END e;
```

Figure 6. Sample code to establish a model hierarchy

In the hierarchy in Figs. 6 and 7, *c* has both *b* and *a* as *ancestors* in the refinement hierarchy. *d* also has *a* as an ancestor, but it is a refinement which is *incompatible* with *b* and *c*. We use inheritance in ASCEND for two reasons. **First**, it is efficient to be able to state that a model contains all the code used to define another. One avoids rewriting the repeated code. This use is not the important reason, however.
Second, and much more important, is that we know that models *b* and *c* are everything model *a* is and more; thus any model that has a part of type *a* in it can use a part of type *b* or type *c* as the actual part when we compile the model.

Changing the type of a part is called *deferred binding* (using the IS_REFINED_TO statement) and is what we believe is one of the important modeling concepts required in equational-based modeling.

A common mechanism for accomplishing deferred binding without explicitly using the IS_REFINED_TO statement occurs when we configure using merging (ARE_THE_SAME). The output stream from a reactor may be a liquid stream with the Unifac model defining its thermodynamic properties. We know the exit stream type within the context of the reactor as that is what the reactor produces. On the other hand, we have created the flash unit with the intention that it can accept any type of stream as an input. Thus we only require that it is a molar stream. If the liquid stream model is a refinement of molar_stream (as it is in our library defining stream types), then, when we

effect the merge, the ASCEND system compiler will make the stream into a model of the more refined type before it does anything else in the process of merging.

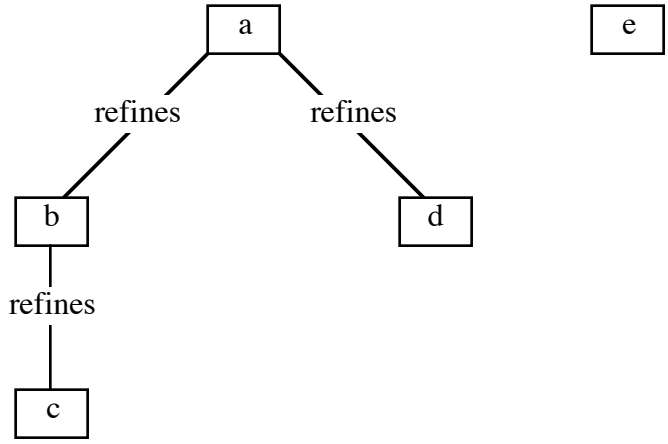


Figure 7. Diagram of the model hierarchy implied by code in Fig. 6

Running an ASCEND model through the interactive interface

Setting up the ASCEND system

We solve an ASCEND model by starting up the ASCEND interface. You should first set up the ASCEND system on the ASCEND *host machine* (i.e., the machine into which you telnet to run ASCEND) by adding the code in Fig. 8 to your .login file on that machine. It is also useful to set an alias into your .cshrc file on that machine to point to the script that starts the ASCEND system, as shown in Fig. 9. It is assumed that you have telnetted into the ASCEND *host machine* from *your local computer*. The instructions to accomplish this are shown in Fig. 10. These instructions assume that you wish to start the emacs editor and then the ASCEND system.

ASCEND works best with the x-windows manager MWM. Some of its functionality is lost with other managers; for example, clicking on the buttons in the toolbox opens the named tool kit in MWM, which it fails to respond for some of the other managers.

We also organize our directory structures on the host machine as follows. Add a subdirectory called *ascend* to your home directory. Then open that subdirectory and add two subdirectories called *libraries* and *examples* respectively. Put all your examples into the *examples* subdirectory and any *libraries* you create into the libraries subdirectory. Your subdirectory structure will now match that used by the main ascend file structure and that used by most other users.

```
setenv ASCENDEDITOR path_to_ascend_editor
setenv ASCENDDIR /usr/local/ascend
setenv ASCENDLIBRARY $ASCENDDIR/libraries:$ASCENDDIR/examples
setenv ASCENDSCRIPT $ASCENDDIR/examples
setenv PATH {$PATH}:{ASCENDDIR}/ghostscript
setenv GS_LIB $ASCENDDIR/ghostscript:$ASCENDDIR/ghostscript/fonts
setenv ASCENDWITHSETS 1
```

Figure 8. Code to be included in .login file to set the environment variables for using ASCEND. The names in italics in the first statement is to be filled in for the particular computer system on which ASCEND is installed.

```
alias ascend /usr/local/ascend/interfacesSX/bin/ascend.bin
```

Figure 9. Alias command to install into .cshrc file so ascend system can be started up simply by issuing the command ascend

```
xhost +name_of_ascend_host_machine
telnet name_of_ascend_host_machine
setenv DISPLAY name_of_your_local_computer:0.0
cd ~/ascend/examples
emacs&
ascend
```

Figure 10. Typical commands to start up the *emacs* editor and then the ASCEND system from a remote terminal

- (Optional) In the editor e-macs, open the file simple_fs.asc by executing

  ctrl-x ctrl-f

  and give the file name

  /usr/local/ascend/examples/simple_fs.asc

  Explore the file and note that it contains the same models as we have given in Figs. 1, 3, 4 and 5 earlier.

- Click on the button labeled SCRIPT in the toolbox to open the SCRIPT tool kit if it is not already opened.

- Using the LHS mouse button, click on Edit in that tool kit and then on Read file. In the window that opens, type

  simple_fs.s

  which will load the script that you can use to load, compile and solve the models in the file simple_fs.asc. ASCEND finds this model automatically by looking

first in your default directory and then in the directories pointed to with the instructions that should be added to the .login file in Fig. 8.

- Using the LHS mouse button, highlight the first two instructions and execute them by clicking on *Execute* in the header of the Script tool kit and then on *Statements selected*. ASCEND loads the model (simple_fs.asc) into the library tool kit.

Example 1: Running the simple flowsheet model

Running a script

We shall solve the simple flowsheet example that is defined in Figs. 4, 5 and 6. Start the ASCEND system as described above. The system will post a number of windows on your screen, the first of which is the ASCEND toolbox. It has a first row of buttons labeled *dir*, *printer* and *exit* and a second row of buttons labeled *LIBRARY, SIMS, BROWSER, SOLVER, PROBE, UNITS, DISPLAY* and *SCRIPT*. These latter set of buttons identify the various tool kits available through the interface. It might be a good idea to open up each tool kit and move the window to a convenient location on your screen.

- Using the LHS mouse button, click on the *dir* button in the toolbox if you wish to point to any particular directory that ASCEND should first look into for any files you ask it to load. You will find it has set the default path to the directory in which you started up the ASCEND system.

- Continue with the next instruction in the script and observe what happens as it executes. ASCEND will compile an instance of *test_controller*, calling it *tc*.

- Select *tc* in the Simulations tool kit (opened by clicking on SIMS), click on *Export* and then *to BROWSER*. You have just exported the compiled model to the BROWSER where you can look at it.

- Run the next instruction in the script which runs the values procedure in tc. To display this procedure, return to the LIBRARY tool kit, pick the model test_controller, and under the Display tool set pick Code (i.e., click in turn on *LIBRARY* in the toolbox, on *test_controller* in the right window in the LIBRARY tool kit, on *Display* in the tool sets listed at the top of the LIBRARY tool kit and finally on *Code* in the window that drops down).

  You can see that the procedures runs the values procedure for fs first. (Can you get the code for this procedure to display? Note first that fs IS_A test_flowsheet. Pick in turn *test_flowsheet*, *Display* and *Code* in the LIBRARY tool kit.) Almost the entire model is the values procedure for it. Note that the contents of the DISPLAY can be scrolled. For convenience, you can also enlarge the DISPLAY window and move it to a more convenient position.

- Return back to the BROWSER. Click on the grill-like window at the top (just to the right of the word BROWSER). Select *display atom values* and then *OK*. ASCEND will now display the values for all variables and constants rather than their type.

- Pick any item listed in the right hand window of the BROWSER - e.g., fs. Note ASCEND places it into the LHS window and makes it the current object. Continue exploring the entire model. Does it make sense to you?

- Prove that the model is "wired" up correctly. Pick *tc* as the current object by clicking on it in the LHS window. Next pick *fs*, then *m1*, then *feed*, then *[2]*. The current object is now tc.fs.m1.feed[2]. Under *Find* (third tool set at top of BROWSER), pick *Aliases*. The window that opens displays two different names. Both these names are for the current object. The first says that the second feed to the mixer m1 is also the second output stream from the splitter sp1. Is this right?

  Try showing that the entire flowsheet is correctly wired up by looking at all the input and output streams to all the units in tc.fs.

- To see how the values procedure is picked through the interface, make tc the current object. Then under *Edit* pick *Initialize*. A window opens listing the available procedures you can pick to run. If you do not see a values procedure, did you make tc.fs the current object? Pick *values* and *OK* (or *Cancel* as the values procedure already executed).

- Make the current object *tc*.

- Under *Export*, pick *to SOLVER*. The SOLVER window opens (move it to a convenient spot). Read the information displayed at the bottom of the window. This model has 43 relations, of which 43 are equalities which are included and none are inequalities. It has 55 variables and all are mentioned in at least one of the 43 included relations. All 55 of the variables are free to be computed (their fixed flags are equal to FALSE).

  The last information entry says this model is *under* specified.

- Run the reset procedure for tc (either by returning to the SCRIPT or by returning to the BROWSER and finding and executing it). Run it from the SCRIPT to be sure you make no errors this time. The numbers change immediately in the SOLVER window, indicating now that 12 of the variables are now fixed, leaving 43 which are free. The system is square - it has 43 equations in 43 variables.

- Solve the model by running the next instruction in the script. Alternatively, you can pick the *Execute* tool set in the SOLVER and under that the *Solve* tool.

- Explore the model in the BROWSER, examining the values for all the variables.

Changing what is fixed and resolving

At this point you are ready to have some real fun with ASCEND. You can reset values for some of the fixed variables and resolve. You can change the status of a variable from fixed to free or vice versa and solve again. The following instructions will lead you through doing this latter type of analysis.

- Make tc.fs.fl1.liq.f['c'] the current object in the BROWSER.

- Note that its fixed attribute is false (RHS window in the BROWSER). That means ASCEND computed it. Let's try fixing this variable and then altering its value and resolving.

  Under the *Edit* tool set, pick *Fix variable.* Because the model is well-posed already, ASCEND drops a window listing a number of currently fixed variables, one of which you need to release (make computable) to make the model well-posed again. Enlarge this window or scroll it to see all the options. Pick *tc.conv* and then *OK* in that window. Note that ASCEND has altered the fixed flag for tc.fs.fl1.liq.f['c'] to true.

- Under the *Edit* tool set, pick *Set value*. The value 0.43194 should be displayed. Edit it to a nearby value, say 0.42.

- Resolve the model. The model will very quickly resolve.

Using the PROBE

By now it should be evident that you would like to monitor some of the variable values while changing others. You can use the PROBE tool set to aid in this monitoring. Try the following.

- Pick the current variable in the BROWSER, tc.fs.fl1.liq.f['c'] and export it to the PROBE. Also export tc.conv. (As time progresses, you can change which variables are in the PROBE. Note there are five different PROBE collections which you can construct for display.)

- Pick the first variable displayed in the PROBE (yes, click on it). Export it to the BROWSER where it now becomes the current object.

- Alter its current value to 0.4 (Edit, Set value). Note it immediately has this value by watching the PROBE.

- Note the value of tc.conv and then solve the model. Note how tc.conv changes from 0.067256 to 0.062832.

Clearing and starting again

Okay, now you can do a lot of playing with this model. Try anything you would like to do. You cannot hurt the model nor the ASCEND system. If all else fails, you can start again by doing the following.

- Open SIMS.

- Select tc and, under the *Edit* tool set, *Delete* it.

- Return to the SCRIPT and run all the instructions starting with the COMPILE instruction (the third one).

  Alternatively you can recompile it yourself by opening the *LIBRARY*. Under *Create*, pick *Compile.* The system offers you the name tc (you can edit this name

to anything else). Hit *OK*. The SIMULATION tool kit opens with the compiled object tc. Export to the BROWSER and, if you like, to the SOLVER.

Putting variables values in the spreadsheet

ASCEND can put values into a spreadsheet so you can see them all and manipulate them. The following assumes you just ran the script from the COMPILE instruction to the last one.

- In the SOLVER, under Display, execute the Spreadsheet tool.
- Expand the window to see the entire set of variables.
- The spreadsheet uses Lotus 123 like instructions. Hit /q to quit the spreadsheet.

Debugging tools

There is a debugging tool in ASCEND which allows you to examine the solving activity in more detail.

Altering SOLVER attributes

- Click on the radiator like icon next to the word SOLVER. A list of adjustable attributes for the SOLVER will display. You can alter any of these. For example, altering the detailed solving info. required attribute will cause ASCEND to output lots of output into the window in which you started the ASCEND system.

  Any of the attributes can be edited in this window.

Running the debugger tool

- In the SOLVER, under *Analyze*, pick *Debug*. You can now pick a variable, an equation, or a block of equations in the system and examine it.

Find dependent equations

In the SOLVER under analyze is the tool to find dependent equations. *Find dependent eqns*. looks for equations that are causing the problem to be numerically singular. It lists a locally linear equation that a set of the equations is satisfying; e.g., it might tell you that

eqn. 12 = 3* eqn. 10 - 0.5*eqn. 2.

It only tells you the equation numbers so you must use the debugger tool (see just above) to discover which equations are involved. If the same equations are continually showing up in such a locally linear relationship, then you can become very suspicious that one of them is redundant; i.e., there is very likely a way to derive one of them from the others algebraically. It is particularly easy if the coefficients are small integers.

Other *Analyze* tools

Under the Analyze tool set in the SOLVER are three other tools that can aid in debugging. *Over-specified* analyzes the SOLVER object and produces a list of variables, one of which you must release for the system to be square. If more than one has to be released, you must release one by picking one on the list and then reanalyze using *Over-specified* again.

Evaluate unincluded eqns. evaluates the residuals for all the equations which were not included while solving and reports those which are not satisfied.

Other tools available

Many other tools exist in ASCEND. There is no problem with exploring each. The worst that happens is that you have to delete the current simulation from the system, recompile and start again.

Example 2: Executing a library model

We shall now try a much more difficult task using ASCEND: picking a model in the ASCEND library and testing it. The model is simple_column found in the file

/usr/local/ascend/libraries/column.lib

A complete listing of this library is also in the document entitled ASCEND Libraries. You should have one or both versions showing this model available to read before proceeding.

We proceed as follows.

- First we will need to create a test model for the model *simple_column*. We open a new file in the editor, calling it "**test_simple_column.asc**" in which we shall construct this model.

There is no documentation to read about this model. All our information about the model is in the ASCEND code itself. So this exercise is a test of our ability to read this code and make sense of it. Looking at the code we first note that the *simple_column* model is a refinement of a *tray_stack* model. We see that the *tray_stack* model is in the same library; we should also look at it.

In *simple_column*, we see a variable called *feed_loc* which is of type *integer*. We guess this to mean the tray number onto which we place the feed.

The next few statements refer to the trays, starting with tray[1] which is refined to be a *condenser*. It appears we are numbering from the top of the column down. The tray pointed to by *feed_loc* is refined to be a *simple_feed_tray*, supporting our earlier conjecture. The tray pointed to by *ntrays* is refined to be a *reboiler*, and all other are set to be of type *simple_tray*. So a simple column has a single feed tray. We guess that it has products withdrawn from only the condenser and reboiler as the rest are simple trays. Overall, that seems logical.

Who defined the variable *ntrays*? It is not defined in this model so it must be a part of the model tray_stack. We can check and discover that it is. We also note the existence of the set called *components*. It is of type string. It must be a list of the components in the column.

We can now start to construct the test routine. We place the following statements into the file **test_simple_column.asc**. The procedures are there because we are following the good style this document strongly advocates.

```
MODEL test_simple_column;

   c                                    IS_A  simple_column;
   c.ntrays                             := 10;
   c.feed_loc                           := 5;

   c.components                         := ['A','B','C'];

   INITIALIZE

      PROCEDURE clear;
         RUN c.clear;
      END clear;

      PROCEDURE specify;
         RUN c.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset:

      PROCEDURE values;
      END values;

      PROCEDURE scale;
      END scale;

END test_simple_column;
```

Writing the values procedure

Remember that the procedure called values has to provide values for all the fixed variables for the model. We have not finished it as we do not know which variables the author of this model (Bob Huss) decided to fix. We could guess, but that looks hopeless (and it is). We could also try to trace recursively through the specify procedures.

Trying that, we see that the *simple_column* specify procedure first runs its own *seqmod* procedure. Okay, so we switch our attention to that procedure. It first runs the *seqmod* procedures for all the trays. One by one we need to examine the models for the trays. The top tray is a condenser. Where is its model?

We do not find it in this library, but, at the top of this library, Bob has listed all the libraries this library needs. One is called *flash.lib*. Looking in there we find the

condenser definition. (Our sanity is beginning to slip and our stomach is churning. This approach is not looking very promising, but we persist.)

We locate the *condenser* model (we see it refines the *VLE_flash* model). We find its *seqmod* procedure. The first thing this procedure does is run the *specify* procedure for VLE. Huh? What is VLE? It is not in this model. Oh, yes, it could be in the *VLE_flash* model definition. At this time, we look around the room and suggest that it is time for coffee.

Of course, we headed down this path here only to show how hopeless it is. There must be a better way.

Finding things by type

Why not let ASCEND tell us which variables Bob has fixed? Can it? There is a tool in the *Find* tool set within the **BROWSER** tool kit that is called *By type*. In other words, the name suggests we can find things by type. To find by type means that we can locate any part within the compiled model by describing its attributes. We want to find all the variables (all are of base type generic_real) in the compiled model which have their fixed flag set to true.

We need to find out about the tool *By type*. There is a description of it in the document entitled **ASCEND ChangeLogs**. This set of logs is a record of all the modifications Peter Piela has made to the ASCEND system. It is also found on-line in the directory

/usr/local/ascend/changeLog

Looking in v16.mail (available on-line), we find a message dated Jul 18 17:39 EDT 1993. In this message Peter describes how to use this tool.

We want to find variables which are fixed. The instructions in the v16.mail tell us that we must first construct a file describing the types of things we wish to locate. Peter offers the following as examples that may be useful.

```
'Find generic_real'  : generic_real;
'Find fixed'         : generic_real WHERE fixed;
'Find eligible'      : generic_real WHERE eligible;
```

Proceding as follows, we shall now use ASCEND to help us write the values procedure for our test model.

- Using the editor, put the statements shown just above into a new file and save it in our directory under the name "**find.file**"
- Start the ASCEND system (if already in the ASCEND system, return to the **LIBRARY** tool kit and run the instruction *Delete all types* under the *Edit* tool set.)
- Open the **SCRIPT** tool kit and set it to record instructions (push the button in on that instruction under the grilled window icon at the top and then push *OK*).

- Next return to the **LIBRARY** tool kit and load the following libraries:

  atoms.lib
  components.lib,
  thermodynamics.lib
  stream.lib
  flash.lib
  plot.lib
  column.lib

  (Note, these are listed at the beginning of the **column.lib** description as being required for using the **column.lib**.) If you have been watching the **SCRIPT** tool set, you will note that the system is writing a script for you which will allow you to repeat all these steps using it later.

- Load our newly created file "**test_simple_column.asc**."

- Next compile **test_simple_column**, calling the compiled object "tsc."

- In the tool kit **SIMS**, select tsc and export it to the **BROWSER**.

- In the **BROWSER** under the *Edit* tool set, select Initialize. Select *reset* and then *OK*.

- Export tsc to the **SOLVER**. The **SOLVER** reports the model is square. It has 349 variables of which 303 are free variables (to be computed using the 303 included equalities) and 45 are fixed.

- At this point, turn off the record instruction in the **SCRIPT** tool kit. (Push the button under the grill at the top;it will pop up. Hit *OK*..)

- Back in the **BROWSER**, run the tool *By type* in the *Find* tool set. The Find by type window will open. Open the file

  ~/find.file

  which we created just above. The three find by type definitions load. Pick *find fixed* and then hit the *Find* button. The **Matches** window opens listing the 45 variables which are fixed.

- Select all the variables in this window (scroll to the top, using the left mouse button, select the top entry on the list, scroll to the bottom, hold the Shift key and select the bottom entry - all entries should now be highlighted). Export these to the **PROBE**.

The **PROBE** contains all the variables that Bob fixed for his model. The instruction

tsc.c.trays[1].alpha['a'] = 1.00000

shows up for all trays and for all species (it is a good idea to double check that it shows up for ALL trays and ALL species). We can compactly add instructions in the **values**

procedure of our test model to set the relative volatilities for all the species on all the trays. The instructions:

```
c.trays[1..c.ntray].alpha['A'] := 4.0;
c.trays[1..c.ntray].alpha['B'] := 2.5;
c.trays[1..c.ntray].alpha['C'] := 1.0;
```

do that. Note, that the tsc part of the instruction is removed as we are adding this instruction into our test model of which tsc is a compiled instance. Also, note that we must refer to *ntrays* as *c.ntrays*. Not calling it *c.ntray* is perhaps the most common error we shall make when writing these instructions.

As we account for setting the value for each variable, it is convenient to delete it from the **PROBE**. We delete a variable from the **PROBE** by picking the variable and using the *Remove selection* instruction under the *Edit* tool set. We shall continue until we have looked at all variables listed in the **PROBE**.

The instruction

```
tsc.c.trays[6].cmo_ratio =  1.00000
```

shows up for only trays 2, 3, 4, 6, 7, 8 and 9. Bob apparently has set it for all the simple_trays. We need to know what this variable is before we can decide how to set it. (One way is to call up Bob Huss and ask him (412-268-5212). Another is to play detective on the models and try to find where it is defined. Returning to the document listing the libraries, we look for the model **simple_tray.** It is located in **flash.lib**. This model defines *cmo_ratio* as a factor (a dimensionless real) and uses it in what looks like a material balance. If we set *cmo_ratio* to one, the material balance says the total liquid flow into the tray equals the total liquid flow leaving. Ah, cmo must stand for constant molar overflow!

So why is this variable there? Why not just write that the total liquid molar flow into the tray equals the total liquid molar flow leaving? Perhaps ours is not to understand. On the other hand, we know that the use of a constant molar overflow assumption replaces a heat balance for a stage. A plausible answer is that Bob wants his simple_column model to be more general. For example, if he adds a heat balance to the equations defining the tray, then the liquid flow in would not generally equal that out. To allow this to happen, he could add in the heat balance (taking away a degree of freedom for the model) and then give it back by making *cmo_ratio* a computed variable.

We set the *cmo_ratio* for all the *simple_trays* to 1.0.

```
c.trays[2..c.feed_loc-1].cmo_ratio              := 1.0;
c.trays[c.feed_loc+1..ntrays-1].cmo_ratio       := 1.0;
```

tsc.c.reduce is more of a mystery. Bob really should add a comment on what it is. (Shame on him, right?) On asking him, he uses it to aid in numerically closing the heat balance for a tray when one is included in later refinements of the tray models. Supposedly it can be set to any value desired here.

```
c.reduce := 1.0;
```

We notice next that the *reflux_ratio* is fixed. We see that it is a variable within the first tray which we know is a *simple_condenser*. Looking at the *simple_condenser* model in **flash.lib**, we see that *reflux_ratio* has the standard definition. The reflux in the column (the stream liquid['liquid']) has a total flow equal to the *reflux_ratio* time the distillate top product flow. When looking at that equation, we note it mentions a possible vapor product.

In the list of fixed variables in the **PROBE**, we see next one called

tsc.c.trays[1].vle.phi['vapor'].

Is this somehow related to the permitted vapor product at the top? Believing it is, we set it to zero (when the model solves, we must check out this conjecture).

```
c.trays[1].vle.phi['vapor'] := 0.0;
```

We see that the distillate total liquid flow out is fixed. This makes sense. It is a common specification for a column. We see also that the feed flows for each of the species are all fixed. Again, that makes sense. We add statements to specify them using a rule of thumb noted earlier that a normal sized chemical process has a feed flow about 1 kg/s. A typical molecular rate is 100 kg/kg_mole so we can have a feed about the normal size if we use the values for the flow of about 0.01 {kg_mole/s}.

```
c.trays[feed_loc].input['feed'].f['A'] := 0.01 {kg_mole/s};
c.trays[feed_loc].input['feed'].f['B'] := 0.01 {kg_mole/s};
c.trays[feed_loc].input['feed'].f['C'] := 0.01 {kg_mole/s};
```

We would like our column to produce A at the top and B and C at the bottom, so we can now choose a value for the distillate top flow. To separate A from B and C, one would set the distillate top flow to the molar flow of A in the feed. The distillate will not be pure as the column is not infinite in size nor will we use an infinte reflux ratio, but this value is the best setting for its flow.

```
c.trays[1].liqout['distillate'].ftot :=
            c.trays[feed_loc].input['feed'].f['A'];
```

Next we spot a variable called *q* which is set for tray 5. The thermal quality of the feed to a column is usually called *q*. *q* is the fraction of the feed which joins the liquid stream going down the column; *1-q* is the fraction that joins the vapor stream the textbooks on distillation. We look around for the definition of *q*. It is referenced as

tsc.c.trays[5].q

which makes it a variable belonging to the type *simple_feed_tray* (which tray[5] is). Locating *simple_feed_tray* in **flash.lib**, we see it introduces *q* and uses it in a material balance exactly the reverse of what we expected. (Bob will have to revise this model in the next version of the library.) Anyway, we set *q* to 0.0 so all the feed will be treated as saturated liquid.

The final variable we need to specify has the name

tsc.c.trays[10].vapsplit['vapor_product']

which will again require some educated guessing. It seems to be the vapor split for the vapor product in the reboiler. We look at the reboiler model in **flash.lib**. It is not there. The *reboiler* model refines the*VLE_flash* model; we look at it (again, in the **flash.lib**). We find the variable about a third of the way into the model definition (somewhat before the *INITIALIZE* section). It is defined as an array over the set *vapout*. A few statements later it is used in an equation. Close examination of the equation suggests it is a material balance. It seems to be the fraction of the vapor produced in the flash that exits in a particular vapor output stream. The stream is called *vapor_product* here; we want no vapor product from the reboiler, so we set this value to zero.

The meanings for some of these variables are at best conjectures on our part at this time. We will take our chances that we have them right. The final *values* section for our model becomes:

```
PROCEDURE values;

   c.reduce                                    := 1.0;
   c.trays[1].reflux_ratio                     := 2.0;
   c.trays[1].vle.phi['vapor']                 := 0.0;

   c.trays[2..c.feed_loc-1].cmo_ratio          := 1.0;
   c.trays[c.feed_loc+1..c.ntrays-1].cmo_ratio
                                               := 1.0;

   c.trays[1..c.ntrays].alpha['A']             := 4.0;
   c.trays[1..c.ntrays].alpha['B']             := 2.5;
   c.trays[1..c.ntrays].alpha['C']             := 1.0;

   c.trays[c.feed_loc].q                       := 0.0;
   c.trays[c.feed_loc].input['feed'].f['A']    := 0.01 {kg_mole/s};
   c.trays[c.feed_loc].input['feed'].f['B']    := 0.01 {kg_mole/s};
   c.trays[c.feed_loc].input['feed'].f['C']    := 0.01 {kg_mole/s};

   c.trays[1].liqout['distillate'].ftot        :=
         c.trays[c.feed_loc].input['feed'].f['A'];

   c.trays[c.ntrays].vapsplit['vapor_product']     := 0.0;

END values;
```

(This version is the result of attempting to run it and then removing the syntax errors from it. Even with experience, we made all sorts of errors (like calling it *feed_loc* instead of *c.feed_loc*).)

- Using the editor, add these statements to the definition of our model *test_simple_column* in the file **test_simple_column.asc**.

Solving the model

- Remove just the model called *test_simple_column* from the **LIBRARY** tool kit by selecting it (in the right window) and running *Delete selected type* under the *Edit* tool set. ***t*est_simple_column.asc** remains in the left window, but we have removed its only model.

- In the tool kit **SIMS**, delete *tsc*.

- Return to the **SCRIPT**, pick the last three instructions and execute them (READ FILE test_simple_column.asc, COMPILE tsc, and RUN tsc.reset).

- Start recording in the **SCRIPT** again.

- Return to the **BROWSER** and run the values procedure for the model. Export *tsc* to the **SOLVER** and solve. If all was done correctly, it will solve quickly (ours did when we tested it).

Checking conjectures on the meanings of the variables

Now would be a good time to verify all the conjectures about the meaning of the variables. We can check the following by browsing the model.

1. Is there a vapor product from the condenser and, if so, does it have a zero flow rate as we wanted?

2. Is there a vapor product from the reboiler and, if so, does it also have a zero flow as we wanted?

3. Does the liquid flowing down the column increase in flow rate as it passes by the feed tray by an amount equal to the feed flow rate (0.03 {kg_mole/s})? It should.

4. Does altering the value of the variable *tsc.c.reduce* seem to alter any of the solution values? It is not supposed to if our understanding is correct.

5. Is the column wired up as expected?

We can interactively modify the types of the parts included in this model, incrementally converting it into a rigorous column model which uses nonideal thermodynamic models to evaluate the physical properties of the three species. To accomplish this latter conversion, we have to associate the species with actual ones in the **components.lib**. The interactive incremental compile statements are included in the **BROWSER** under the *Edit* tool set.