CHAPTER 17 A SIMPLE CHEMICAL ENGINEERING FLOWSHEETING EXAMPLE

In this example we shall examine a model for a simple chemical engineering process flowsheet. The code listed below exists in the file in the ASCEND examples subdirectory entitled *simple_fs.asc*. Except for some formatting changes to make it more presentable here, it is exactly as it is in the library version. Thus you could run this example by loading this file and using it and its corresponding script *simple_fs.s*.

17.1 THE PROBLEM DESCRIPTION

This model is of a simple chemical engineering flowsheet. Studying it will help to see how one constructs more complex models in ASCEND. Models for more complex objects are typically built out of previously defined types each of which may itself be built of previously defined parts, etc. A flowsheet could, for example, be built of units and streams. A distillation column could itself be built out of trays and interconnecting streams.

Lines 40 to 56 in the code below give a diagram of the flowsheet we would like to model. This flowsheet is to convert species B into species C. B undergoes the reaction.

B-->C

The available feed contains 5 mole percent of species A, a light contaminant that acts as an inert in the reactor. We pass this feed into the reactor where only about 7% of B converts per pass. Species C is much less volatile than B which is itself somewhat less volatile than A. Relative volatilities are 12, 10 and 1 respectively for A, B and C. Species A will build up if we do not let it escape from the system. We propose to do this by bleeding off a small portion (say 1 to 2%) of the B we recover and recycle back to the reactor.

The flowsheet contains a mixer where we mix the recycle with the feed, a reactor, a flash unit, and a stream splitter where we split off and remove some of the recycled species B contaminated with species A

Our goal is to determine the impact of the bleed on the performance of this flowsheet. We would also like to see if we can run the flash unit to get us fairly pure C as a bottom product from it.

The first type definitions we need for our simple flowsheet are for the variables we would like to use in our model. The ones needed for this example are all in the file atoms.a4l. Thus we will need to load atoms.a4l before we load the file containing the code for this model.

The following is the code for this model. We shall intersperse comments on the code within it.

17.2 THE CODE

As the code is in our ASCEND models directory, it has header information that we require of all such files included as one large comment extending over several lines. Comments are in the form (* comment *).

To assure that appropriate library files are loaded first, ASCEND has the REQUIRE statement, such as appears on line 61:

```
REQUIRE atoms.a4l
```

This statement causes the system to load the file *atoms.a4l* before continuing with the loading of this file. *atoms.a4l* in turn has a require statement at its beginning to cause *system.a4l* to be loaded before it is.

```
(*********************************************************************\      1
                        simple_fs.asc                                        2
                        by Arthur W. Westerberg                              3
                        Part of the Ascend Library                           4
                                                                             5
This file is part of the Ascend modeling library.                            6
                                                                             7
Copyright (C) 1994                                                           8
                                                                             9
The Ascend modeling library is free software; you can redistribute          10
it and/or modify it under the terms of the GNU General Public License as    11
published by the Free Software Foundation; either version 2 of the          12
License, or (at your option) any later version.                             13
                                                                            14
The Ascend Language Interpreter is distributed in hope that it will be      15
useful, but WITHOUT ANY WARRANTY; without even the implied warranty of      16
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU           17
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.

Use of this module is demonstrated by the associated script file
simple_fs.s.
\*******************************************************************)

(*******************************************************************\
  $Date: 97/02/20 18:54:21 $
  $Revision: 1.5 $
  $Author: mthomas $
  $Source: /afs/cs.cmu.edu/project/ascend/Repository/models/examples/simple_fs.asc,v $
\*******************************************************************)
(*

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



                                          -------
                                         |       |
                   ----------------------| split |----> purge
                  |                      |       |
                  |                       -------
                  |                          ^
                  v                          |
                -----      ---------      -------
               |     |    |         |    |       |
         ----->| mix |--->| reactor |--->| flash |
               |     |    |         |    |       |
                -----      ---------      -------
                                             |
                                             |
                                              ----->  C

This model requires: “system.a4l”
                     “atoms.a4l”
*)

REQUIRE atoms.a4l
```

The first model we shall define is for defining a stream. In the document entitled “Equation-based Process Modeling” we argue the need to define a stream by maximizing the use of intensive variables and the equations interrelating them. Our problem here requires only the molar flows for the components as the problem definition provides us with all the physical properties as constants. Nowhere for this simple model do we seem to need temperatures, fugacities, etc. To maximize the use of intensive variables, we will use mole fractions and total molar flow to characterize a stream. We must include the equation that says the mole fractions add to unity. Our first model we call *mixture*.

```
(* ************************************************* *)                        62
                                                                             63
MODEL mixture;                                                               64
                                                                             65
   components                     IS_A set OF symbol_constant;               66
   y[components]                  IS_A fraction;                             67
                                                                             68
   SUM[y[i] | i IN components] = 1.0;                                        69
                                                                             70
METHODS                                                                      71
   METHOD clear;                                                             72
      y[components].fixed := FALSE;                                          73
   END clear;                                                                74
                                                                             75
   METHOD specify;                                                           76
      y[components].fixed := TRUE;                                           77
      y[CHOICE[components]].fixed := FALSE;                                  78
   END specify;                                                              79
                                                                             80
   METHOD reset;                                                             81
      RUN clear;                                                             82
      RUN specify;                                                           83
   END reset;                                                                84
                                                                             85
END mixture;                                                                 86
                                                                             87
```

Line 66 of the model for mixture defines a set of symbol constants. We will later include in this set one symbol constant giving a name for each of the species in the problem (A, B and C). Line 67 defines one mole fraction variable for each element in the set of components, while line 69 says these mole fractions must add to 1.0.

We add a methods section to our model to handle the flag setting which we shall need when making the problem well-posed -- i.e., as a problem having an equal number of unknowns as equations. We first have a method called clear which resets all the “fixed” flags for all the variables in this model to FALSE. This method puts the problem into a known state (all flags are FALSE). The second method is our selection of variables that we wish to fix if we were to solve the equations corresponding to a mixture model. There is only one equation among all the mole fraction variables so we set all but one of the flags to TRUE. The CHOICE function picks arbitrariliy one of the members of the set *components*. For that element, we reset the fixed flag to FALSE, meaning that this one variable will be computed in terms of the values given to the others.

The reset method is useful as it runs first the clear method to put an instance of a mixture model into a known state with respect to its fixed flags, followed by runing the specify method to set all but one of the fixed flags to TRUE.

These methods are not needed to create our model. To include them is a matter of modeling style, a style we consider to be good practice. The investment into writing these methods now has always been paid back in reducing the time we have needed to debug our final models.

The next model we write is for a stream, a model that will include a part we call *state* which is an instance of the type mixture.

```
(* ************************************************* *)

MODEL molar_stream;

   components     IS_A set OF symbol_constant;
   state          IS_A mixture;
      Ftot,f[components]   IS_A molar_rate;

   components, state.components     ARE_THE_SAME;

   FOR i IN components CREATE
      f_def[i]: f[i] = Ftot*state.y[i];
   END;

METHODS

   METHOD clear;
      RUN state.clear;
      Ftot.fixed  := FALSE;
      f[components].fixed:= FALSE;
   END clear;

   METHOD seqmod;
      RUN state.specify;
      state.y[components].fixed:= FALSE;
   END seqmod;

   METHOD specify;
      RUN seqmod;
      f[components].fixed:= TRUE;
   END specify;

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

   METHOD scale;
      FOR i IN components DO
         f[i].nominal := f[i] + 0.1{mol/s};
      END;
      Ftot.nominal := Ftot + 0.1{mol/s};
   END scale;

END molar_stream;

```

We define our stream over a set of components. We next include a part which is of type mixture and call it *state* as mentioned above. We also include a variable entitled *Ftot* which will represent the total molar flowrate for the stream. For convenience -- as they are not needed, we also include the molar flows for each of the species in the stream. We realize that the components defined within the part called *state* and the set of components we just defined for the stream should be the same set. We force the two sets to be the same set with the ARE_THE_SAME operator.

We next write the equations that define the individual molar flows for the components in terms of their corresponding mole fractions and the total flowrate for the stream. Note, the equations that says the mole fractions add to unity in the definition of the state forces the total of the individual flowrates to equal the total flowrate. Thus we do not need to include an equation that says the molar flowrates for the species add up to the total molar flowrate for the stream.

We again write the methods we need for handling flag setting. We leave it to the reader to establish that the specify method produces a well-posed instance involving the same number of variables to be computed as equations available to compute them. The scale method is there as we may occasionally wish to rescale the nominal values for our flows to reflect the values we are computing for them. Poor scaling of variables can lead to numerical difficulties for really large models. This method is there to reduce the chance we will have poor scaling.

Note that the nominal values for the remaining variables -- the mole fractions -- are unity. It does not need to be recomputed as unity is almost always a good nominal value for each of them.

Our next model is for the first of our unit operations. Each of these will be built of streams and equations that characterize their behavior. The first models a mixer. It can have any number of feed streams, each of which is a molar stream. We require the component set for each of the feed streams and the output stream from the unit to be the same set. Finally we write a component material balance for each of the species in the problem, where we sum the flows in each of the feeds to give the flow in the output stream, *out*.

```
(* ************************************************* *)

MODEL mixer;

   n_inputs                  IS_A  integer_constant;
   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;

METHODS

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

   METHOD seqmod;
   END seqmod;

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

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

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

END mixer;
```

The *METHOD clear* sets all the fixed flags for the parts of this model to false by running each of their clear methods (i.e., for all the feeds and for the stream out). If this model had introduced any new variables, their fixed flags would have been set to FALSE here.

We will implement the method to make the model well posed into two parts: *seqmod* (stands for "sequential modular" which is the mindset we use to get a unit well-posed) and *specify.* The first we shall use within any unit operation to fix exactly enough fixed flags for a unit such that, if we also make the feed streams to it well-posed, the unit will be well-posed. For a mixer unit, the output stream results simply from mixing the input streams; there are no other variables to set other than those for the feeds. Thus the *seqmod* method is empty. It is here for consistency with the other unit operation models we write next. The *METHOD specify* makes this model well-posed by calling the *seqmod* method and then the *specify* method for each of the feed streams. No other flags need be set to make the model well-posed.

METHOD reset simply runs *clear* followed by *specify.* Running this sequence of method will make the problem well-posed no matter which of the fixed flags for it are set to TRUE before running *reset*. Finally, flowrates can take virtually any value so we can include a *scale* method to scale the flows based on their current values.

The next model is for a very simple 'degree of conversion' reactor. The model defines a turnover rate which is the rate at which the reaction as written proceeds (e.g., in moles/s). For example, here our reaction will be B --> C. A turnover rate of 3.7 moles/s would mean that 3.7 moles/s of B would convert to 3.7 moles/s of C. The vector stoich_coef has one entry per component. Here there will be three components when we test this model so the coefficients would be 0, -1, 1 for the reaction

```
0 A + (-1) B + (+1) C = 0.
```

Reactants have a negative coefficient, products a positive one. The material balance to compute the flow out for each of the components sums the amount coming in plus that created by the reaction.

```
(* ************************************************* *)

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_Afactor;

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

METHODS

   METHOD clear;
      RUN feed.clear;
      RUN out.clear;
      turnover.fixed                      := FALSE;
      stoich_coef[feed.components].fixed  := FALSE;
   END clear;

   METHOD seqmod;
      turnover.fixed                      := TRUE;
      stoich_coef[feed.components].fixed  := TRUE;
   END seqmod;

   METHOD specify;
      RUN seqmod;
      RUN feed.specify;
   END specify;

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

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

END reactor;

```

The *METHOD clear* first directs all the parts of the reactor to run their *clear* methods. Then it sets the fixed flags for all variables introduced in this model to FALSE.

Assume the feed to be known. We introduced one stoichiometric coefficient for each component and a turnover rate. To make the output stream well-posed, we would need to compute the flows for each of the component flows leaving. That suggests the material balances we wrote are all needed to compute these flows. We would, therefore, need to set one fixed flag to TRUE for each of the variables we introduced, which is what we do in the *METHOD seqmod*. Now when we run *seqmod* and then the *specify* method for the feed, we will have made this model well-posed, which is what we do in the *METHOD specify*.

The flash model that follows is a constant relative volatility model. Try reasoning why the methods attached are as they are.

```
(* ************************************************* *)

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;

METHODS

   METHOD 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;

   METHOD seqmod;
      alpha[feed.components].fixed     := TRUE;
      vap_to_feed_ratio.fixed          := TRUE;
   END seqmod;

   METHOD specify;
      RUN seqmod;
      RUN feed.specify;
   END specify;

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

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

END flash;

(* ************************************************* *)
```

The final unit operation model is the splitter. The trick here is to make all the states for all the output streams the same as that of the feed. This move makes the compositions all the same and introduces only one equation to add those mole fractions to unity. The rest of the model should be evident.

```
MODEL splitter;

   n_outputs                    IS_A  integer_constant;
   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;

METHODS

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

   METHOD seqmod;
      split[1..n_outputs-1].fixed:=TRUE;
   END seqmod;

   METHOD specify;
      RUN seqmod;
      RUN feed.specify;
   END specify;

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

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

END splitter;
```

```
(* ************************************************* *)

```

Now we shall see the value of writing all those methods for our unit operations (and for the models that we used in creating them). We construct our flowsheet by saying it includes a mixer, a reactor, a flash unit and a splitter. The mixer will have two inputs and the splitter two outputs. The next few statements configure our flowsheet by making, for example, the output stream from the mixer and the feed stream to the reactor be the same stream.

The methods are as simple as they look. This model does not introduce any variables nor any equations that are not introduced by its parts. We simply ask the parts to clear their variable fixed flags.

To make the flowsheet well-posed, we ask each unit to set sufficient fixed flags to TRUE to make itself well posed were its feed stream well-posed (now you can see why we wanted to create the methods *seqmod* for each of the unit types.) Then the only streams we need to make well-posed are the feeds to the flowsheet, of which there is only one. The remaining streams come out of a unit which we can think of computing the flows for it.

```
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;

  METHODS

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

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

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

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

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

END flowsheet;

(* ************************************************* *)

```

We have created a flowsheet model above. If you look at the reactor model, we require that you specify the turnover rate for the reaction. We may have no idea of a suitable turnover rate. What we may have an idea about is the conversion of species B in the reactor; for example, we may know that about 7% of the B entering the reactor may convert. How can we alter our model to allow for us to say this about the reactor and not be required to specify the turnover rate? In a sequential modular flowsheeting system, we would use a "computational controller." We shall create a model here that gives us this same functionality. Thus we call it a "controller." There are many ways to construct this model. We choose here to create a model that has a flowsheet as a part of it. We introduce a variable conv which will indicate the fraction conversion of any one of the components which we call the key_component here. For that component, we add a material balance based on the fraction of it that will convert. We added one new variable and one new equation so, if the flowsheet is well-posed, so will our controller be well-posed. However, we want to specify the conversion rather that the turnover rate. The *specify* method first asks the flowsheet fs to make itself well-posed. Then it makes this one trade: fixing conv and releasing the turnover rate.

```
MODEL controller;

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

METHODS

   METHOD clear;
      RUN fs.clear;
      conv.fixed:=FALSE;
   END clear;

   METHOD specify;
      RUN fs.specify;
      fs.r1.turnover.fixed:=FALSE;
      conv.fixed:=TRUE;
   END specify;

   METHOD reset;
      RUN clear;
      RUN specify;
   END reset;

   METHOD scale;
      RUN fs.scale;
   END scale;

END controller;

(* ************************************************* *)
```

We now would like to test our models to see if they work. How can we write test for them? We can create test models as we do below.

To test the flowsheet model, we create a test_flowsheet model that refines our previously defined flowsheet model. “To refine the previous model” means this model includes all the statements made to define the flowsheet model plus those statements that we now provide here. So this model is a flowsheet but with it components specified to be ‘A’, ‘B’, and ‘C’. We add a new method called *values* in which we specify values for all the variables we intend to fix when we solve. We can also provide values for other variables; these will be used as the initial values for them when we start to solve. We see all the variables being given values with the units specified. The units must be specified in ASCEND. ASCEND will interpret the lack of units to mean the variable is unitless. If it is not, then you will get a diagnostic from ASCEND telling you that you have written a dimensionally inconsistent relationship.

Note we specify the molar flows for the three species in the feed. Given these flows, the equations for the stream will compute the total flow and then the mole fractions for it. Thus the feed stream is fully specified with these flows.

We look at the seqmod method for each of the units to see the variables to which we need to give values here.

```
MODEL test_flowsheet REFINES flowsheet;

   m1.out.components:==[‘A’,’B’,’C’];

  METHODS

   METHOD values;
      m1.feed[1].f[‘A’]          := 0.005  {kg_mole/s};
      m1.feed[1].f[‘B’]          := 0.095 {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;
      r1.turnover                  := 3 {kg_mole/s};

      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;

(* ************************************************* *)

```

Finally we would like to test our controller model. Again we write our test model as a refinement of the model to be tested. The test model is, therefore, a controller itself. We make our fs model inside our test model into a test_flowsheet, making it a more refined type of part than it was in the controller model. We can do this because the test_controller model is a refinement of the flowsheet model which fs was previously. A test_flowsheet is, as we said above, a flowsheet. We create a values method which first runs the values method we wrote for the test_flowsheet model and then adds a specification for the conversion of B in the reactor.

```
MODEL test_controller REFINES controller;

   fs       IS_REFINED_TO  test_flowsheet;
   key_components :=='B';

METHODS

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

END test_controller;

(* ************************************************* *)

```