Design and Use of Dynamic Modeling in ASCEND IV[1]

Jennifer L. Perry

Benjamin A. Allan

ABSTRACT[2]

We present an alternative design for communicating information between an interactive, equation-based modeling environment user and an initial value problem (IVP) solver. The design is practically demonstrated by connecting the widely available integrator LSODE and the freely available plotting program Xgraph to the ASCEND IV[3] modeling environment. In contrast to the ASCEND III integrator interface, this new design allows an order of magnitude improvement in the ease of developing an understanding of a dynamic model's behavior. The improvement is in part achieved by allowing the user to interactively define what time varying quantities should be recorded for postprocessing by graphics packages or other tools without requiring that a new ASCEND model be compiled, thus making visualization easy. The more significant improvement, however, is the new ability to select any part of the complete dynamic model and solve it, thus making exploration of component dynamic behaviors easy. Finally, the user may interactively change a variable between algebraic and dynamic (state or derivative) roles. This makes it possible to negotiate a solution to the index problems which frequently arise in chemical engineering.

---

1. This work has been supported by the Engineering Design Research Center, a NSF Engineering Research Center.
2. Comments or questions on this report should be addressed to ascend+bb@cs.cmu.edu and ballan@cs.cmu.edu.
3. The ASCEND IV software is available free (on the publication date) via http://www.cs.cmu.edu/afs/cs/project/ascend/home/Home.html for any UNIX platform with X11 and ANSI C.

Table of Contents

1 INTRODUCTION

We believe a reusable chemical engineering model is a general model of some unit operation or process phenomenon which may be used to model steady-state and dynamic flowsheet problems. Moreover, it may be solved without extensive initialization work in each new application. From this definition it follows that ASCEND III [PMW93] does not have reusable models within its libraries. However, with ASCEND IV, we come closer to obtaining reusable dynamic models in the ASCEND environment. We will address reusability and the difficulties in ASCEND III that make it nearly impossible to reuse a dynamic model in Section 2 and Section 4. We keep several design goals, including obtaining reusability, in mind while exploring solutions to this problem in ASCEND IV. We do not want to increase the compile time of the models or slow the ASCEND interface, and there must be a clear migration path for models in ASCEND III to ASCEND IV. We detail the design goals in Section 3.

The reuse of existing ASCEND dynamic models is impossible due primarily to the way that ASCEND III communicates problem specifications to the integration package, LSODE. (An example of this communication is given in Section 9). Reusability is also hampered by the way that ASCEND III communicates integration results to the interactive user. By having the two separate integrator interface models, **lsode** and **derivatives**, as well as the separate output collecting **plt_plot** model, ASCEND III leads to repetition of code, time consuming and error prone FOR loops, and extreme inconvenience in flowsheeting. We can solve these problems by identifying the observation, derivative, and state variables at run time by setting flags that are new parts of the basic variable type **solver_var**. The flags enable us to create a dynamic model which is entirely self-contained, thus we can build up a dynamic model from smaller dynamic models. We can use the larger model to study the dynamics of both the parts and the whole using a single compiled object. This increases flowsheeting capabilities and the readability of models, while decreasing compile time. Because ASCEND is object-oriented, this addition of flags to the **solver_var** does not affect existing libraries of purely algebraic models.

We pose other solutions to the human/computer communication problem and explain the design difficulties that make them infeasible in Section 5.1. We explain the new **solver_var** flags and their use in detail in Section 5.2 and introduce a new plotting tool, ASC_PLOT, in Section 5.2.1. ASC_PLOT is interactive and eliminates the need for a separate ASCEND plot model. Although the total solution presented in Section 5 brings us closer to reusability, we do not have "full" reusability in ASCEND IV.

In Section 6 we review some of the remaining problems. The thermodynamics library and the stream library pose problems when trying to create very general dynamic models because of difficulties with the existing type hierarchy. However, with the addition of the **CASE** statement (a feature never implemented in ASCEND III) in ASCEND IV, we expect to solve these problems in future development.

In Section 8 we present a simple example of a dynamic flash drum modeled, solved, and plotted in ASCEND IV. In Section 9 we present the most complete and detailed example available from ASCEND III, an ammonia synthesis flowsheet [Mon96]. A more detailed description of the new ASCEND IV plotting capability is available in the ASCEND IV on-line help and may be given in a future technical report.

A brief note on typography may help in understanding this report. The names of types, known as classes in many other object-oriented (OO) languages, are written in **bold.** The ASCEND keywords are in capitalized **BOLD**. The names of compiled objects (the OO term is "instances") and of parts defined within a type (in which case the object is to be compiled mentally by the reader) are in italicized ***bold***. The names of ASCEND methods, which can only be executed as parts of an instance, are also written in italicized ***bold***. Fragments of code offset from the text are in Courier type. It should be clear from the context whether the fragment in question is C, Tcl, or ASCEND.

2 WHAT IS REUSABILITY?

The ideal library in ASCEND IV would consist of models which are reusable. By reusable, we mean that each model in the library should represent an accurate mathematical description of a general physical system and all its time variant as well as steady-state behaviors. We must be able to reuse such models in a larger context without doing extensive configuration or initialization of them. This representation should be general enough so that it can handle multiple conditions arising within the system, for example, the appearance and disappearance of chemical species or thermodynamic phases.

Most particularly, we should be able to put a reusable model into a flowsheet with other reusable models to describe an entire process without any changes to the definitions of the individual models. This implies that the models must be entirely “self-contained”. All the attributes needed to solve the individual model must exist in one concise definition so that essentially all we need to do is “connect” the models by a series of short commands. For

example, the flowsheet fragment shown here should be modeled from library definitions **mixer**, **reactor**, and **flash** as follows:

```
MODEL fragment;
     m IS_A mixer;
     r IS_A reactor;
     f IS_A flash;
     m.outstream, r.instream ARE_THE_SAME;
     r.outstream, f.instream ARE_THE_SAME;
END fragment;
```

This way we can solve the difficult “degrees of freedom” problem that occurs in dynamic modeling using the same simple techniques advocated by [Wes96] for strictly algebraic systems built of complex, self-contained parts. According to this criterion, reusability does not exist in the published dynamic models in ASCEND III.

3 DESIGN GOALS

We strongly believe that, particularly where research code is concerned, documenting the rationale leading to an artifact (the code) is as important as documenting the artifact itself. In the next three sections we will attempt to document both. In this section we state the goals fo the design process in a necessarily fuzzy fashion.

While developing *reusable* models in the ASCEND IV environment, it is important for us to keep several other design goals or criteria in mind. Although it is our aim to have models which will be general enough to describe many physical processes, it is important that the method we choose to meet this goal does not increase the consumption of computer resources dramatically. Moreover, any changes made in the ASCEND C source code should not drastically slow down the user interface.

Along with these issues of minimizing compile time, solution time and result analysis time comes a need for a solution which will not require us to rewrite a lot of the C source code. Therefore, the solution to creating reusable models must fit into the general scheme of existing user interface and solver C code. To make the C source code readable and "elegant", it is desirable for the new source code to be "symmetric" with the existing code. That is, C functions for managing the differential-algebraic problem to be solved should be the same as or very similar to the functions used to manage a purely algebraic problem. These functions should also be easy to maintain.

Finally, because there are so many models which already exist in the libraries of ASCEND III, the solution must not invalidate these pre-existing models. Ideally, we do not want to have to rewrite the ASCEND III model libraries to incorporate the ASCEND IV solution. With these goals in mind, we will approach the problem in a systematic, engineering design manner, examining in more detail the failures in ASCEND III to better understand what is needed in ASCEND IV and documenting several potential solutions.

4 PROBLEMS WITH REUSABILITY IN ASCEND III

By looking at previously published dynamic simulation models in the ASCEND III libraries, we quickly see why these models are not reusable models. It is essentially an impossible task to solve these models without an extensive lesson in dynamic modeling and initialization or a complete understanding of the phrase "degrees of freedom" because they cannot be simply connected and because they do not have standard methods for managing degrees of freedom and initialization.

We cannot simply connect these models because of the multiple models used to communicate information to the ASCEND III integration and plotting tools. In order to represent and integrate one physical system, we need four separate models. Two of the interface models, **plt_plot** and **lsode**, contain large arrays of time sampled information which is unreadable to users except in graphical form. We discover a modeling method which brings us closer to reusability and relieves us of the difficulties of using these several models. We also suggest (by a detailed example in Section 8) a style of creating standard methods to handle steady-state and dynamic degrees of freedom. The creation of such methods, however, must be done specifically for each model in a library by the library author.

4.1 THE MATHEMATICAL PROBLEM

The mathematical problem that we are typically attempting to solve in ASCEND is a system of differential-algebraic equations (DAE)

$$dy[i]/dt = gd[i](y,z,t) \tag{1}$$

$$0 = ga(y,z,t) \tag{2}$$

where i = number of equations. The mathematical system contains a derivative variable (dy/dt) which possibly is a vector of derivatives, a state variable (y), and an independent variable (t). There are generally also algebraic equations *(ga)* and variables *(z)* constraining the states and derivatives in some way. The equations may even be tangled so that the semi-explicit form given above cannot be easily obtained, for example in variable volume systems.

The integration package that ASCEND III uses is LSODE [RH93], which solves initial value problems for systems of first-order differential equations. It obtains the ODE systems from an ASCEND interface model in the following form:

$$f(i) = f(t, y[1], y[2], .., y[neq], dy[1]/dt,...,dy[neq]/dt) \tag{3}$$

where $i = 1..neq$ and at every evaluation point ASCEND solves and eliminates the algebraic equations completely. There is the implicit assumption in LSODE that *y[i]* and *dy[i]/dt* are the i^{th} state-derivative pair.

If we were to write the ODEs found in ASCEND simulation models in this form it would definitely create many problems in readability and naming. For example, it would be hard to understand whether *dy(1)_dt = yin - yout* was an energy balance, a mass balance, or a momentum balance. Therefore, the format in which ODEs are written in ASCEND models (e.g. ***dMtot_dt* = *Min* - *Mout* + *Mrxn***) must be converted to the equation (3) information which LSODE can interpret, but a user need not see.

4.2 lsode AND derivatives IN ASCEND III

In ASCEND III, conversion of ODE information is accomplished by setting up two different models with special part and type names which an IVP solver interface can interpret. These integrator interface models are **derivatives** and **lsode**. The user must then refine these models to accommodate the DAE found in their simulation model and merge each derivative and state variable to its corresponding part in the models **derivatives** and **lsode** using the **ARE_THE_SAME** operator. Because each state variable and its derivative partner must have similar array locations, this conversion technique results in a series of cumbersome, nested **FOR** loops for all but the most trivial dynamic models. Moreover, when using this method of communicating with LSODE, we must create four different type definitions to effectively use a dynamic model:

- Code describing the physical system through a set of differential-algebraic equations.
- Code refining the integrator interface model **derivatives**.
- Code refining the integrator interface model **lsode**.
- Code refining the plot interface model **plt_plot** which at compile time defines specific output variables to be recorded and graphed.

It is here we see that the problem of keeping dynamic models self-contained lies not within the LSODE integration package, but with how ASCEND III transmits the integration information to LSODE. By using the idea of two different integrator interface models, when we write a flowsheet (e.g. a chemical process with three unit operations), models have to be written for the three units and then two overall models would have to refine the **derivatives** and **lsode** model for the flowsheet. Finally, a model would have to be written to plot the flowsheet.

Using this method, the individual units within the flowsheet cannot be integrated separately because each model does not contain the information needed to communicate with LSODE. This is in no way close to being reusable.

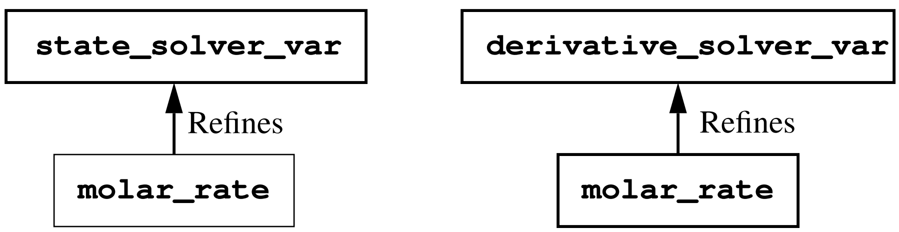
- To integrate, the models require an extensive amount of interface overhead modeling by the user.
- The user cannot simply “connect” unit operations together, separate flowsheet models must be rewritten for each different combination of unit operations.

One purpose of modeling is to test different combinations of unit operations rapidly without conducting expensive experiments. Here we do not save much time by having to rewrite three models for each new combination of units! It is almost as though the ASCEND III interface just described was designed to waste user time. In Section 9 there is an example of the ASCEND III integrator interface model refinements associated with an ammonia reactor, written by Jon Ingar Monsen. By looking at these models, it is easy to see why these models make this ammonia reactor unit operation close to impossible to insert into another flowsheet without rewriting a lot of code.

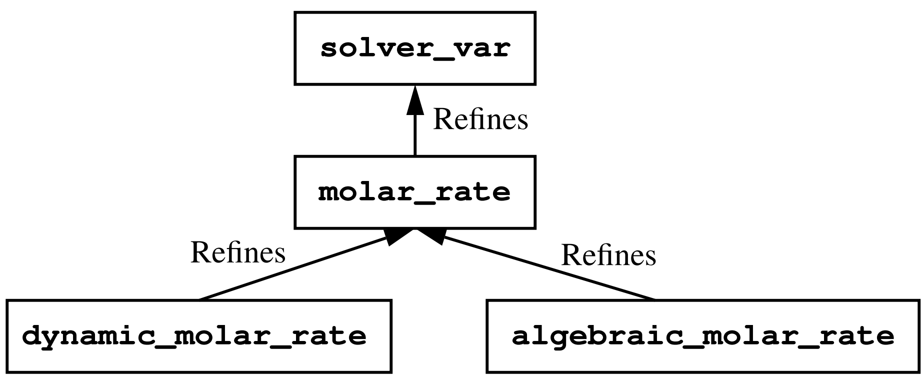
5 REUSABILITY IN ASCEND IV

The question we now ask is "How can we interface LSODE without creating four separate models?" We came up with three possible solutions, one of which we accepted and implemented. The others were rejected because of the reuse problems they posed.

- Identify the derivative and state variables at run time by having flags within the **solver_var** [AW97] that users can manipulate interactively, as is done for communicating algebraic problem specifications to ASCEND III's algebraic solvers.
- Identify states and derivatives by giving each type a different root in the type hierarchy.



- Identify states and derivatives by recognizing prefixes in type names.



5.1 REJECTED SOLUTIONS

One of the solutions to the integration interface problem that we rejected was to create multiple **solver_var** types. The multiple types would be named: **state_solver_var**, **derivative_solver_var**, and **algebraic_solver_var**. LSODE would then be programmed to recognize the difference in these **solver_var** types and treat the variables which were associated with them accordingly when integrating. There are many problems with this solution. The main one was that we would have to rewrite all ASCEND III models to accommodate the separation of the **solver_var**. Also, the problem of matching a state variable and its corresponding derivative variable would still exist. Furthermore, we would have to create multiple *atoms.lib* because **solver_var** is at the root of the ASCEND IV hierarchy, unless this hierarchy was totally reorganized. This solution is also inflexible because the role of a state or derivative variable may change to an algebraic variable in other dynamic configurations.

Another solution that we considered, but abandoned was creating atom refinements with the special name prefixes **dynamic_** and **algebraic_**. However, this solution also would cause type incompatibility within ASCEND IV (i.e. prevent merging) and again require multiple versions of *atoms.lib*.

5.2 FLAGS WITHIN Solver_var

By flagging the derivative, state, and independent variables, we are able to accomplish the goal of keeping the models self-contained. This is done by creating new **solver_var** children called ***ode_type***, ***ode_id***, and ***obs_id***. ***Ode_type*** determines the variable's role with respect to an IVP solver.

Table 1: Definition of *Ode_type* values

ode_type value	variable type
-1	independent
0	algebraic
1	state
2	derivative
>2	higher derivatives

The function of ***ode_id*** is to pair the state variable with its corresponding derivative variable (e.g. ***Mtot.ode_id*** = 1 and ***dMtot_dt.ode_id*** = 1). This allows LSODE to match the correct variables during integration. The ***ode_id*** is a prototype way of associating variables that will become obsolete when ASCEND IV differential calculus syntax is implemented. ***Obs_id*** allows us to select the variables we wish to record while integrating.

The different values for the newly created **solver_var** children are set in the following ways. Values of ***ode_type*** and ***ode_id*** are determined in the user-written ***set_ode*** method which is in the **METHODS** section of the dynamic model. This method only has to be created once for each model. If the model is put into a larger hierarchical model, we simply run the ***set_ode*** method recursively for the individual sub-models, as is done with the standard ***reset*** method [Wes96]. In addition to setting the ***ode_type*** within the **METHODS** section of the model, we can change this instance interactively through the ASCEND interface. This is beneficial when negotiating an index problem. The ***ode_type*** has a default of zero so that we do not have to go through and set it for all of the algebraic variables, which can amount to tens of thousands variables when modeling large systems. The only variables which require assignment of the ***ode_type*** value are the derivatives, the states, and the independent variable.

While developing the **solver_var** child ***ode_id***, we needed to develop a universal counter, which we called **ode_counter**. This counter is needed because it is important that there is a distinct value of the ***ode_id*** for each matching derivative and state variable. The type **ode_counter** is a **UNIVERSAL ATOM** that refines **integer**. It is found in *atoms.lib* in ASCEND IV. To set the ***ode_id*** in the **METHODS** section of the simulation model, we simply use this **UNIVERSAL ATOM** and define a counter instance of it in the model, usually named ***ode_counter***. We then increment the universal ***ode_counter*** value each time it is used. The advantage of using a universal counter over using specific integer values defined in each dynamic model is seen when the model is put in a flowsheet. The values for ***ode_id*** can be assigned over the entire flowsheet and are not specific to the number or order of the unit operations within the flowsheet.

The only disadvantage to the counter is that it must be set to a value of one each time a new flowsheet configuration is specified. This is accomplished in the script file that accompanies the model by inserting the command:

```
ASSIGN {<model name>.ode_offset} 1 {};
```

One might ask why the counter does not just default to a value of one. Setting a default would defeat the purpose of a “universal” counter. We want the counter to start at a value of one for the top model we select interactively and then continue to count throughout the model hierarchy below this top model. This may not include the whole flowsheet, therefore, we want to be able to interactively control when the universal counter is reset to one.

The ***obs_id*** flag is an integer, but it may be thought of as a boolean by the casual user. All variables with ***obs_id*** > 0 when the integrator is started will be recorded in tabular form in a file called *obs.dat*. The state and derivative variables are automatically recorded in the file *y.dat*. The file names *y.dat* and *obs.dat* may be respecified interactively through the ASCEND solver interface. These files can be fed into the new plotting tool in ASCEND IV, ASC_PLOT, which will be discussed in Section 8. The ***obs_id***s > 0 are reassigned to a unique positive value by the interface to LSODE for file indexing purposes. Thus, we need not create hierarchical methods for uniquely indexing the observations as we do for the ***ode_id***.

All variables have a default ***obs_id*** value of zero. Therefore, if we want to observe a variable, we set the variable's ***obs_id*** to one in the ***set_obs*** method in the **METHODS** section of the test model. In using this method of assignment, it becomes apparent that in large flowsheets a method which would "clear" all observations is needed. This is accomplished through the ***clear_obs*** method in the individual models. It sets all ***obs_id*** values back to zero.

5.2.1 ASC_PLOT

Many, if not most, numerical results are most easily understood by creating plots of the data rather than looking at individual values, lists of values, or averages of values. We focus in ASCEND IV on producing data in a portable text format carrying all the information possibly needed by the range of plotting tools found at Carnegie Mellon. We do this because each organization or user has a favorite plotting tool set and because the present interests of the authors do not include research on data visualization. There are many graphing tools available within spreadsheets on personal computers, as stand-alone utilities (Xgraph, XMGR), or as complete graphing environments (gnu-plot, Yorick) on UNIX workstations. The use of an ***obs_id*** makes this particularly easy, indeed it is the foundation of an improved plotting ability.

The local users of ASCEND are not content with a raw data set, however, so we also provide a Tcl/Tk [Ost94] based tool, ASC_PLOT, for manipulating data files and creating plots with Xgraph or any other plot tool which can accept a text file as input. This tool is similar to an independent spreadsheet package in that after the simulation is finished it allows the user to:

- import arbitrarily large amounts of data from ASCEND or from other sources.
- execute elementary mathematical transformations on rows or columns of the data.
- view selected portions of the data in graphical formats he or she can interactively configure.
- merge data sets from several simulations

This tool does none of these things as well as a dedicated PC package such as EXCEL might. However, nobody will grant us degrees in Chemical Engineering for writing a spreadsheet or a graph package. ASC_PLOT is a Tcl/Tk application that does not require the rest of ASCEND IV to be loaded.

In ASCEND III, a separate plot type definition and instance must be created, as mentioned in Section 4.2. This requires defining fixed size arrays of plot points associated with specific observation variables. All of this costs compiler time and a great deal of ASCEND object overhead memory just to store what are essentially vectors of real numbers. Furthermore, the observed variables cannot be dynamically redefined and cannot be tracked across multiple runs in one plot. Clearly, ASCEND III style programming is not the ideal method to handle the large quantities of data produced using dynamic models.

5.2.2 Changes Within the Script File

Some changes that accompanied this solution occurred within the models' script file. As was mentioned earlier, the universal counter's initial value must be assigned within the script. Another addition is that the independent variable's initial value must be specified within the script file or another method. This done by the following command:

```
ASSIGN {<independent variable name>} 0 {s};
```

This specific command assigns zero as the initial point for integration. The {s} is units expression of the independent variable, seconds. The independent variable may be in any units, however, as appropriate to the model.

The command

```
WRITE_VIRTUAL <model name> <buffer name>;
```

saves the current values of variables to a core memory buffer. If there is a problem with integration, the buffer contents can be used to reinitialize the model.

We also had to add to the script file a method to set the observation sample time steps. We developed a Tcl command called *set_int*. In order to access this command, we load the file *set_intervals.tcl* which is now a part of the ASCEND IV examples library. *Set_int* is invoked by the command:

```
     set_int <number of steps> <step size> <units>,
e.g. set_int 101 1 {s}
```

This means “create 1 second intervals (implicitly from t=0) for 100 seconds”. *Set_int* replaces the method within the ASCEND III model **lsode**. Variations on the *set_int* method are easy to construct, and several are included in the library files. For example, *set_lagrangeint* is a function which creates uniformly spaced major intervals with minor intervals at the roots of a scaled lagrange polynomial.

Finally, to begin integration, all the user must type in the script file is:

```
INTEGRATE <model name>;
```

The indices to a subset of the defined intervals may also be given, e.g.:

```
INTEGRATE <model name> FROM 50 TO 100;
```

Some of these changes within the script file should eventually be moved into the methods of the models or into new graphic user interface tools. This would remove extraneous information from the eye of the user, making it easier to work with dynamic models.

5.2.3 Advantages to Flagging Variables

There are many advantages to the solution above. By flagging the variables, we take away the need for the two separate integrator interface models, **lsode** and **derivatives**, as well as the separate plotting model. In terms of flowsheeting, this allows us to connect unit operations by simply merging the outputs of one with the inputs of another. This simpler means of connection, combined with the notion of setting ***ode_type***s and ***ode_id***s in the methods of individual models, allows us to do something we previously could not, namely solve and integrate individual models and parts of a flowsheet. There is a sharp increase (from 0) in the reusability of models, because they are now self-contained.

Being able to choose observation variables interactively by setting the ***obs_id*** value to one through the interface increases the model's flexibility. If we only want to see the dynamics of one variable, we do not have to record extraneous variables specified by the model's original author as is the case in ASCEND III. Moreover, when these observation variables are viewed in ASCEND III, they are found in the BROWSER window. If we want to save these variables for future use, they must be sent to the PROBE and saved from there. In ASCEND IV they are automatically saved to a specified data file. Also, the observation values in ASCEND III are indexed like the dynamic variables in LSODE: ***obs[1..n_obs][1..n_steps]***. This makes it hard to distinguish which variable is which in the BROWSER or PROBE. Now when the values are written to the data file, they are tabulated according to the names that the modeler assigned to them in the dynamic model.

We next consider the notion of “elegant” modeling. One way to describe model code elegance is the absence of extra statements and structures concerned with trivial issues, such as file input and output. These statements contain information other than the mathematical essence of the model. In ASCEND IV the models become more readable because all of the arrays and size parameters which were previously found in the **derivatives** and **lsode** refinements are now derived based on variable flags and managed by the solver interface software. Removing these bookkeeping objects from the model definitions also reduces the compile time of a model.

Another advantage of the new LSODE communication method is that it provides a clear migration path from the simulation models of ASCEND III that do not contain variable flag information about integration. They have **lsode** and **derivative** counterparts which are interfaced with LSODE. The only changes that we need to make within these older models in the ASCEND III libraries is the addition of the ***ode_offset*** and independent variables, and of the methods ***set_ode*** and ***clear_obs*** to handle the flags on the new **solver_var**. Once these variables and procedures have been added and tested under the new system, it is easy to simply delete the old bookkeeping model parts and type definitions. During the transition both integration interfaces may be used with the same dynamic model. Although a lot of source code (in C) had to be written for the ASCEND IV solution, the amount of source coded needed to manage the interface with LSODE is actually smaller in ASCEND IV than in ASCEND III. Once all of the ASCEND III dynamic models have been migrated to ASCEND IV, the ASCEND III LSODE interface C code will be deleted.

5.2.4 Problems Associated with Flagging Variables

One of the problems that we encountered while implementing this solution was what to do about the relative and absolute tolerances that were previously specified in the integrator interface models **lsode** and **derivatives**. We first thought about setting them within LSODE so that they would be internally fixed. However, modelers often need to control the tolerances for their integrations. We have been frustrated on more than one occasion by integrators and other solvers that hide their internal tuning parameters. Another solution which we thought of was to create a model ***tolerance*** which would communicate an array of tolerances to LSODE. However, this brought back the problem of difficult array indexing and inflexibility in changing the role of a variable between algebraic and dynamic. Therefore, the solution that we settled on was adding two more children to **solver_var**: ***ode_atol*** and ***ode_rtol***. Each child is a real and they are set by default to 1e-4 and 1e-8, respectively. This solution yields a symmetric treatment of all the integration interface information: everything LSODE or any algebraic solver needs to know about a variable is contained in flags that are part of the variable. Although the presence of these new flags increases the size of the **solver_var**, the compile time does not dramatically increase.

A solution that we proposed to the problem of a large **solver_var** was to create a CASE-like statement within the **solver_var** that would determine which children to compile based on a universal flag, e.g. DAE = TRUE or FALSE. However, a **CASE** statement does not currently exist for atoms in ASCEND IV. Therefore, this solution is infeasible for the moment. Another alternative solution to the large **solver_var** problem is to allow multiple inheritance for atomic types in ASCEND. This solution we set aside for the moment because the implications of multiple inheritance in the ASCEND language are too complicated to explore here. The solution selected to solve the large **solver_var** problem is to create an alternate *ivpsystem.lib* to replace *system.lib* for dynamic modelers. This way algebraic modelers have an interchangeable library, *system.lib*, that does not contain these larger **solver_var**s.

6 AREAS OF IMPROVEMENT IN ASCEND IV

We do not yet have full reusability in ASCEND IV. Most of the problems which hinder us from accomplishing our goal of reusability are primarily associated with some of the standard libraries and not with the integration package. The libraries that are problematic deal with chemical engineering applications. Currently, the thermodynamics library, which in turn affects the stream library, is separated into two main categories homogeneous and heterogeneous mixtures. When we create a unit operation having streams entering and leaving, the number and type of phases in each stream must be specified to determine the appropriate mixture model if anything more complicated than a mass balance is to be computed. This difficulty particularly affects dynamic models. For example, we cannot simulate the level in a flash tank overflowing into the vapor line because we commit to the overhead line having a **vapor_mixture** properties model when we define the flash tank.

A possible solution to this might come with the **CASE** statement when it arrives in ASCEND IV. With it, we could have a set of equations in the model which would apply to heterogeneous mixtures and a set that would deal with homogeneous mixtures. Then, when the overflow condition occurs, the appropriate liquid equations would be used and the vapor equations ignored. This conditional modeling area is under development by Vicente′ Rico-Ramirez.

Another significant improvement in ASCEND IV would be to attach a more efficient DAE integrator, such as a sparse version of DASSL [BCP89], to the system. Another improvement, and one that is necessary for commercial dynamic modeling, would be to incorporate strategies for handling discontinuities into the integration software and interface. Neither has yet been done because they are not on the critical path of any of the researchers presently working on ASCEND IV.

7 CONCLUSIONS

Through the development of a new integrator interface procedure within ASCEND IV, we advance towards total reusability. When we first looked at the problem of reusability in ASCEND III, we discovered that a large portion of the problem dealt with the way ASCEND III interfaced with the integration package LSODE. Therefore, we developed a new technique to deal with the integration information in a dynamic model. This solution, which consists of flagging the variables within the model, meets the design goals that we established because it:

- Reduces model compilation time
- Does not slow down ASCEND interface
- Does not invalidate pre-existing models
- Makes models more self-contained
- Allows a more “user-friendly” interface for specifying sample times and plotting simulation results
- Leads to better flowsheeting capabilities
- Makes for much more elegant and readable code

Examples of the application of this integration interface solution to chemical engineering unit operations are located in Appendix A. With this solution came the new plotting tool in ASCEND IV, ASC_PLOT, which takes data from observation variables and exports it to third party graphing packages such as Xgraph.

Other solutions to the interface problem that we have discussed were rejected because they did not meet our design criteria. These solutions experienced problems like type incompatibility and the invalidation of existing models. Although the new method of interfacing LSODE brings us closer to reusability, a few areas still hinder us from “full” reusability. One of these problems is the hierarchical structure of the thermodynamics and stream libraries in ASCEND IV. The addition of the **CASE** statement to ASCEND IV is a possible solution to this problem and we would then attain total reusability in ASCEND. Through the development of reusable models, we hope to extend the scope of ASCEND past dynamic and steady state simulation models into the realm of true multi-purpose modeling. This would include the development of models good for dynamic and steady state optimization, as well as parameter estimation.

8 APPENDIX A - ASCEND IV EXAMPLE

8.1 EXAMPLE - FLASH UNIT

A standard example of a chemical engineering unit operation is the flash unit. This separation unit can be described with several levels of thermodynamics, the simplest being relative volatility, which will be the focus of our first example. Without going through the details of the theory behind mathematical modeling, we will describe the general basis of the **alpha_flash** model. The theory which was used to create this model is described in [AW96].

The model presented below is a multi-component flash separation unit where the split fractions, ***alpha[components]***, are defined arbitrarily. The dynamics of the system are described by a component mass balance which is defined for each component in the system. This allows us to track the dynamic variables' behavior in an unsteady state system. The integration is accomplished through the integration package LSODE. The model is connected to this package through the new interface described in prior sections of this paper with the addition of the new methods ***set_ode*** and ***clear_obs***, located in lines 157-173 of the model.

The easiest way to understand the steps used to solve and integrate a model is to go through a script of that model. In lines 217-251 of this appendix, we present a script file for the **alpha_flash** model. The function of this file is to load all of the libraries needed in describing the model, to run the methods setting the values needed to solve the model at steady and unsteady states, and finally to integrate the model at unsteady state. The first set of commands in the SCRIPT file are used to read files into ASCEND IV. Line 217 loads a Tcl file which defines the command ***set_int*** found in line 250. ***Set_int*** sets the observation time samples during integration. The next two commands set aliases, *alib* and *blib*, for frequently used directories so that they may be referenced by the Tcl notation *$alib* or *$blib* when reading libraries into ASCEND IV. Line 220 is used to delete all type definitions and simulations which may have been loaded previously to make the demonstration using a clean system. Finally, lines 221-226 read all of the appropriate libraries needed to define the types within **test_alpha_flash**, along with loading

test_alpha_flash itself.

Before a model can be solved it needs to be compiled as a simulation in ASCEND. In this case, the simulation instance is called ***tf***, however this name is arbitrary. Line 228 handles this action. The next line sends the instance to the BROWSER window so that we can examine the parts of the instance in greater detail. This is useful when we wish to change the initialization values of the variables.

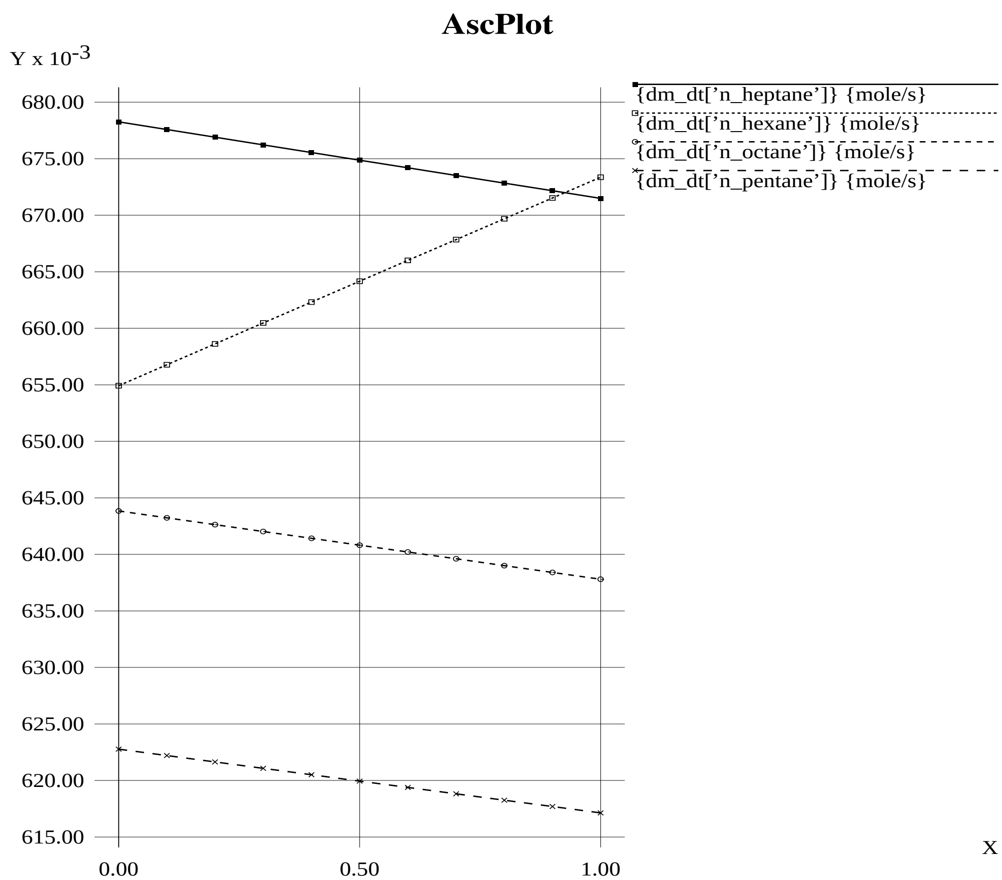
The next set of commands solve **test_alpha_flash** at steady state. We run the methods ***reset***, ***steady_values***, and ***values*** in order to square and initialize the model. Line 235 sends the instance to the SOLVER window and solves the model with the QRSlv.

The fun comes in when we solve the model at an unsteady state and get to play with the derivative variables. This is accomplished in lines 236-251. The first line of this series of commands sets the value of the **UNIVERSAL_COUNTER**, ***ode_offset***, to one. As described before, this counter is used in the matching of the state and derivative pairs. The next task performed is establishing what variables are to be observed during integration. Line 239 clears all variables previously flagged as observation variables. Then line 240 sets them for the variables chosen within the ***set_obs*** method in the test model. If a previous integration has taken place, then time within the model will not start at zero. Therefore we insert line 241 to begin the integration at time equal to zero. The boolean variable ***dynamic*** in the model is set to TRUE in the next line. The model's ***seqmod*** method configures the model for unsteady state use if the value of ***dynamic*** is TRUE, as shown in lines 124-132 of the **alpha_flash** model.

The next three lines square the model (***specify*** calls ***seqmod***) and initialize the values of the model so that it will be ready to solve. We then use the *WRITE _VIRTUAL* <model name> <buffer name>; to save the current values of variables to a core memory buffer. Therefore, if there is a problem with integration, the buffer contents can be used to reinitialize the model. The model is solved with the QRSlv in line 249. Using the *set_int* command that was loaded in line 217, we set the observation time samples. Finally, the model is integrated using BLSODE (the name of the new interface to LSODE in the ASCEND IV environment) from time step 0 to time step 10. We can change these times to whatever time interval we wish to observe.

After the model is solved and integrated we can then use ASC_PLOT to make a graphical representation from the model's output. To do this, we select the ASC_PLOT button in the TOOLBOX window. The data, found in the obs.dat and y.dat files in the current directory, is loaded by selecting "Load data set" in the Edit menu of the ASC_PLOT window. The variables to be graphed are selected and moved into the plotted variables box. After all of the plotted variables are set, we select "View plot file" from the Execute menu and Xgraph will produces a graph, such as the one on the following page.

This is the plot of the accumulation rates of four species in a mass holdup with respect to time, as computed using an **alpha_flash** subject to a step change in its feed composition. The molar rates of holdup change are on the Y axis and time is on the X axis. We can also choose different observation variables to plot through the interactive ASC_PLOT window.



8.2 MODELS USED IN THE FLASH EXAMPLE

```
MODEL alpha_flash;
      components                    IS_A set OF symbol_constant;
      data[components]              IS_A component_constants;
      choice_component              IS_A symbol_constant;
      alpha[components]             IS_A factor;
      inputs, vapouts, liqouts      IS_A set OF symbol_constant;
      input[inputs], vapout[vapouts],
        liqout[liqouts]             IS_A molar_stream;
      scale                         IS_A scaling_constant;
      state                         IS_A heterogeneous_mixture;
      phases                        IS_A set OF symbol_constant;
      phases                        :== ['liquid','vapor'];
      x[components]                 IS_A mole_fraction;
      t                             IS_A time;
```

```
ode_offset                    IS_A ode_counter;

 (*set dynamic to TRUE IF you want the MODEL to be dynamics*)
dynamic                       IS_A boolean;

(*set the local to be TRUE IF the MODEL is being integrated
    within another MODEL*)
local                         IS_A boolean;

liq_frac[components],
    vap_frac[components]IS_A mole_fraction;

Mole_Holdup[phases]           IS_A mole;
M                             IS_A mole;
m[components]                 IS_A mole;
dm_dt[components]             IS_A molar_rate;
Vol_Holdup[phases]IS_A volume;
Vtot                          IS_A volume;
mix_V[phases]                 IS_A molar_volume;

state.reference               :== 'liquid';


(*aliasing*)
components, input[inputs].components,
    liqout[liqouts].components,
    vapout[vapouts].components,
    state.components          ARE_THE_SAME;

phases, state.phases          ARE_THE_SAME;

FOR i IN components CREATE
  alpha[i], state.alpha['vapor'][i] ARE_THE_SAME;
END;

FOR i IN components CREATE
    x[i], state.y[i]    ARE_THE_SAME;
    liq_frac[i], state.mix['liquid'].y[i] ARE_THE_SAME;
    vap_frac[i], state.mix['vapor'].y[i]  ARE_THE_SAME;
END;

state.mix['liquid'], liqout[liqouts].state      ARE_THE_SAME;

state.mix['vapor'], vapout[vapouts].state       ARE_THE_SAME;

FOR i IN components CREATE
    data[i], input[inputs].data[i], state.data[i] ARE_THE_SAME;
END;
```

```
    (*relations*)
    M = SUM(m[i] | i IN components);

    FOR i IN [phases-[state.reference]] CREATE
        Mole_Holdup[i] = M * state.phi[i];
    END;

    M = SUM(Mole_Holdup[i] | i IN phases);
    FOR i IN [components-[choice_component]] CREATE
        M * x[i] = m[i];
    END;

          FOR i IN [phases - [state.reference]] CREATE
        Vol_Holdup[i] = Mole_Holdup[i]*mix_V[i];
    END;

      V_mix_def: state.V = SUM(mix_V[i]|i IN phases);

    V_vhold_def: Vtot = SUM(Vol_Holdup[i] | i IN phases);

    V_def: Vtot = M * state.V;

    FOR i IN components CREATE
        component_MB[i]:  dm_dt[i] = SUM(input[inputs].f[i])-
          SUM(vapout[vapouts].f[i])-SUM(liqout[liqouts].f[i]);
    END;

    (*define bounds*)
    FOR i IN components CREATE
        dm_dt[i].lower_bound := -1e10 {mol/s};
    END;

    local := TRUE;

METHODS
   METHOD clear;
    RUN input[inputs].clear;
    RUN vapout[vapouts].clear;
    RUN liqout[liqouts].clear;
    RUN state.clear;
    m[components].fixed                 := FALSE;
    dm_dt[components].fixed             := FALSE;
    x[components].fixed                 := FALSE;
    M.fixed                             := FALSE;
    Mole_Holdup[phases].fixed           := FALSE;
    Vtot.fixed                          := FALSE;
    Vol_Holdup[phases].fixed            := FALSE;
    dynamic                             := FALSE;

  END clear;
```

```
METHOD seqmod;
  RUN state.specify;
  mix_V[phases].fixed                 := TRUE;
  M.fixed                             := FALSE;
  Vtot.fixed                          := TRUE;
  dm_dt[components].fixed             := TRUE;
  x[components].fixed                 := FALSE;
  liqout['liquid'].Ftot.fixed         := TRUE;
  IF dynamic = TRUE THEN
      dm_dt[components].fixed         := FALSE;
      state.ave_alpha['vapor'].fixed  := FALSE;
      liqout[liqouts].Ftot.fixed      := TRUE;
      vapout[vapouts].Ftot.fixed       := TRUE;
      x[components].fixed := FALSE;
      m[components-[choice_component]].fixed      := TRUE;
      Vtot.fixed                      := TRUE;
      M.fixed                         := FALSE;
      RUN set_ode;
      IF local = TRUE THEN
        input[inputs].state.y[components].ode_type:= 0;
      END;
  END;
END seqmod;

METHOD specify;
  RUN input[inputs].specify;
  RUN seqmod;
END specify;

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

METHOD scale;
  RUN input[inputs].scale;
  RUN vapout[vapouts].scale;
  RUN liqout[liqouts].scale;
  RUN state.scale;
END scale;

METHOD set_ode;
    FOR c IN components DO
      m[c].ode_type := 1;
      dm_dt[c].ode_type := 2;
      m[c].ode_id := ode_offset;
      dm_dt[c].ode_id := ode_offset;
      ode_offset := ode_offset + 1;
  END;
```

```
        t.ode_type:= -1;
    END set_ode;

    METHOD clear_obs;
      FOR c IN components DO
          m[c].obs_id:= 0;
          dm_dt[c].obs_id:= 0;
      END;
    END clear_obs;
END alpha_flash;

MODEL test_alpha_flash REFINES alpha_flash;
      inputs                  :== ['feed'];
      vapouts                 :== ['vapor'];
      liqouts                 :== ['liquid'];
      components  :==
['n_pentane','n_hexane','n_heptane','n_octane'];
      choice_component        :== 'n_hexane';
      data['n_pentane']       IS_REFINED_TO n_pentane;
      data['n_hexane']        IS_REFINED_TO n_hexane;
      data['n_heptane']       IS_REFINED_TO n_heptane;
      data['n_octane']        IS_REFINED_TO n_octane;

  METHODS

    METHOD steady_values;
      M                       := 100 {mole};
      dm_dt[components]       := 0 {mol/s};
      x[choice_component]     := 0.33;
      state.phi['liquid']     := 0.3;
    END steady_values;

    METHOD unsteady_values;
      liqout[liqouts].Ftot          := 10 {mol/s};
      x[components]                 := 1/3;
    END unsteady_values;

    METHOD values;
       input[inputs].f[components]              := 4 {mol/s};
      state.alpha['vapor']['n_pentane']         := 1.5;
      state.alpha['vapor']['n_hexane']          := 1.2;
      state.alpha['vapor']['n_heptane']         := 1.0;
      state.alpha['vapor']['n_octane']          := 1.3;
      state.ave_alpha['vapor']                  := 1.16;
    END values;

    METHOD set_obs;
      FOR c IN components DO
          m[c].obs_id:= 1;
          dm_dt[c].obs_id:= 1;
```

```
      END;
    END set_obs;
END test_alpha_flash;
```

SCRIPT FILE FOR ALPHA_FLASH

```
source /afs/cs.cmu.edu/project/edrc-ascend3/jsed/models/blsode/newbl-
sode/tcl/set_intervals.tcl;
set alib /afs/cs.cmu.edu/project/ascend/newserver/models/pending/librar-
ies;
set blib /afs/cs.cmu.edu/project/edrc-ascend3/jsed/models/blsode/newbl-
sode/newthermo;
DELETE TYPES;
READ FILE $alib/ivpsystem.lib;
READ FILE $alib/atoms.lib;
READ FILE $alib/components.lib;
READ FILE $alib/H_G_thermodynamics.lib;
READ FILE $alib/stream.lib;
READ FILE $blib/flash3.lib;

COMPILE tf OF test_alpha_flash;
BROWSE {tf};

#this solves the flash at steady state#
RUN {tf.reset};
RUN {tf.steady_values};
RUN {tf.values};
SOLVE {tf} WITH QRSlv;

#this solves and integrates the flash at an unsteady state#
ASSIGN {tf.ode_offset} 1 {};
RUN {tf.clear_obs};
RUN {tf.set_obs};
ASSIGN {tf.t} 0 {s};
ASSIGN {tf.dynamic} TRUE;
RUN {tf.specify};
RUN {tf.unsteady_values};
RUN {tf.values};
WRITE_VIRTUAL tf initpoint0;
SOLVE tf WITH QRSlv;
set_int 100 0.1 s;
INTEGRATE tf from 0 to 10 WITH BLSODE;
```

9 APPENDIX B - ASCEND III EXAMPLE

9.1 JON MONSEN'S AMMONIA REACTOR

```
MODEL flowsheet_dynamics REFINES derivatives;

   (* Defines the differential variables *)

   process                    IS_A flowsheet;
   components                 IS_A set OF symbol;
   subset[1..CARD(components)]      IS_A symbol;
   x                                 IS_REFINED_TO time;
   y[1..5*process.n_reactors]       IS_REFINED_TO molar_rate;
   dydx[1..5*process.n_reactors]      IS_REFINED_TO concentration_rate;
   y[5*process.n_reactors+1..6*process.n_reactors] IS_REFINED_TO temper-
ature;
   dydx[5*process.n_reactors+1..6*process.n_reactors]IS_REFINED_TO
temperature_rate;
   components, process.pfr_reactor.components      ARE_THE_SAME;

   subset[1] := 'ammonia';
   subset[2] := 'argon';
   subset[3] := 'hydrogen';
   subset[4] := 'methane';
   subset[5] := 'nitrogen';

   FOR i IN [1..process.n_reactors] CREATE
     FOR j IN [1..CARD(components)] CREATE
       y[j+(i-1)*5],
       process.pfr_reactor.reactor[i].output['product'].f[subset[j]]
ARE_THE_SAME;
       dydx[j+(i-1)*5],
       process.pfr_reactor.reactor[i].dC_dt[subset[j]]ARE_THE_SAME;
     END;
   END;

   FOR i IN [5*process.n_reactors+1..6*process.n_reactors] CREATE
     y[i], process.pfr_reactor.reactor[i-process.n_reactors*5].T
ARE_THE_SAME;
     dydx[i], process.pfr_reactor.reactor[i-process.n_reactors*5].dT_dt
ARE_THE_SAME;
   END;

    n_eq := process.n_reactors*6;

METHODS
   METHOD clear;
      RUN process.pfr_reactor.clear;
      RUN process.heat_ex.clear;
   END clear;
```

```
   METHOD specify;
      RUN process.pfr_reactor.specify_pfr;
      RUN process.heat_ex.specify;
   END specify;

   METHOD reset;
      RUN process.clear;
      RUN process.specify;
   END reset;

END flowsheet_dynamics;


MODEL flowsheet_integrate REFINES lsode;

  (* Connects to the integrator lsode *)

      d                             IS_REFINED_TO flowsheet_dynamics;
      step                          IS_A time;
      npnt                          IS_A integer;
      curve_spacing           IS_A integer;

      nstep := 80;
      npnt := nstep + 1;

      d.n_obs := 6*d.process.n_reactors+1;

      FOR i IN [1..d.n_obs-1] CREATE
        d.obs[i], d.y[i]            ARE_THE_SAME;
      END;

      d.obs[d.n_obs],
        d.process.pfr_reactor.reactor[1].Tf ARE_THE_SAME;

(*
Define the absolute and arelative error tolerance user wants LSODE to
have.
*)
      atol[1..d.n_eq] := 1.0e-5;
      rtol[1..d.n_eq] := 1.0e-5;
(*
Put user suppiled initial conditions into the integration.
*)
      FOR i IN [1..d.n_eq] CREATE
        d.y[i], y[0][i]       ARE_THE_SAME;
      END;

      FOR i IN [1..d.n_obs] CREATE
            d.obs[i], obs[0][i]     ARE_THE_SAME;
```

```
END;

(* Sets up the ploting OF the molar flowrates *)

concentration_plot            IS_A plt_plot_integer;
concentration_plot.ncurve := 5;

FOR i IN [1..concentration_plot.ncurve] CREATE
  concentration_plot.curve[i].npnt := nstep+1;
END;

concentration_plot.curve[1].npnt := nstep+1;
concentration_plot.curve[2].npnt := nstep+1;
concentration_plot.curve[3].npnt := nstep+1;
concentration_plot.curve[4].npnt := nstep+1;
concentration_plot.curve[5].npnt := nstep+1;
concentration_plot.title := '';
concentration_plot.YLabel := 'Molar Flowrate [mole/s]';
concentration_plot.XLabel := 'Time [s]';
concentration_plot.curve[1].legend := 'NH3';
concentration_plot.curve[2].legend := 'Ar';
concentration_plot.curve[3].legend := 'H2';
concentration_plot.curve[4].legend := 'CH4';
concentration_plot.curve[5].legend := 'N2';

FOR i IN [1..concentration_plot.ncurve] CREATE
  FOR j IN [1..concentration_plot.curve[1].npnt] CREATE
    concentration_plot.curve[i].pnt[j].y,
      obs[j-1][(d.process.n_reactors)*4+i] ARE_THE_SAME;
    concentration_plot.curve[i].pnt[j].x,
      x[j-1]                       ARE_THE_SAME;
      END;
END;

(* Sets up the ploting OF the temperatures *)

temperature_plot              IS_A plt_plot_integer;
temperature_plot.ncurve := 6;

FOR i IN [1..temperature_plot.ncurve] CREATE
  temperature_plot.curve[i].npnt := nstep+1;
END;

temperature_plot.title := '';
temperature_plot.YLabel := 'T [K]';
temperature_plot.XLabel := 'Time [s]';

temperature_plot.curve[1].legend := 'T1';
temperature_plot.curve[2].legend := 'T2';
temperature_plot.curve[3].legend := 'T3';
```

```
      temperature_plot.curve[4].legend := 'T4';
      temperature_plot.curve[5].legend := 'T5';
      temperature_plot.curve[6].legend := 'Tf';


      FOR i IN [1..temperature_plot.ncurve-1] CREATE
        FOR j IN [1..temperature_plot.curve[1].npnt] CREATE
          temperature_plot.curve[i].pnt[j].y,
            obs[j-1][(d.process.n_reactors)*5+curve_spacing*i]
ARE_THE_SAME;
          temperature_plot.curve[i].pnt[j].x,
            x[j-1]                              ARE_THE_SAME;
        END;
      END;

       FOR j IN [1..temperature_plot.curve[1].npnt] CREATE
          temperature_plot.curve[temperature_plot.ncurve].pnt[j].y,
            obs[j-1][d.n_obs]            ARE_THE_SAME;
          temperature_plot.curve[temperature_plot.ncurve].pnt[j].x,
            x[j-1]                              ARE_THE_SAME;
       END;

METHODS
   METHOD set_intervals;
      FOR i IN [0..nstep] DO
          x[i] := i*step;
        END;
   END set_intervals;
   METHOD values;
      FOR i IN [0..nstep] DO
            FOR j IN [1..(d.n_eq-d.process.n_reactors)] DO
                  y[i][j].lower_bound := 0.0 {mole/s};
            END;
      END;
      FOR i IN [0..nstep] DO
            FOR j IN [(d.n_eq-d.process.n_reactors+1)..d.n_eq] DO
                  y[i][j].lower_bound := 0.0 {K};
            END;
      END;
      step                          := 5.0 {s};
      x[0]                          := 0.0 {s};
      RUN set_intervals;
      RUN d.process.pfr_reactor.bounds;
   END values;
   METHOD initial_conditions;
       d.process.heat_ex.input['primary'].state.T := 465 {K};
   END initial_conditions;
   METHOD steady_state;
       RUN d.process.reset;
   END steady_state;
```

```
   METHOD clear;
      RUN d.clear;
      y[0..nstep][1..d.n_eq].fixed:= FALSE;
      x[0..nstep].fixed := FALSE;
      step.fixed              := FALSE;
   END clear;
   METHOD specify;
      RUN d.specify;
      RUN fix_x_and_y;
      RUN set_intervals;
      d.process.pfr_reactor.input['feed'].state.T.fixed := FALSE;
      x[0..nstep].fixed := TRUE;
      step.fixed              := TRUE;
   END specify;
   METHOD reset;
      RUN clear;
      RUN specify;
      FOR i IN [1..d.n_eq] DO
        d.y[i].fixed          := TRUE;
      END;
   END reset;
   METHOD set_tols;
      atol[1 ..d.n_eq] := 1.0e-4;
      rtol[1 ..d.n_eq] := 1.0e-4;
   END set_tols;
   METHOD set_init_cond;
      FOR i IN [1..d.n_eq] DO
        y[0][i] := d.y[i];
      END;
      FOR i IN [1..d.process.n_reactors*5] DO
        d.dydx[i] := 0.0  {mole/m^3/s};
      END;
      FOR i IN [d.process.n_reactors*5+1..d.process.n_reactors*6] DO
        d.dydx[i] := 0.0  {K/s};
      END;
   END set_init_cond;

END flowsheet_integrate;
```

10 REFERENCES

[AW96] B. A. Allan and A. W. Westerberg. Equation-based process modeling. Technical Report CAPD short course notes, Engineering Design Research Center, Carnegie Mellon University, http://www.cs.cmu.edu/ascend/doc/C/westerberg/ProcessModeling.ps.Z as of 10/96, 1996.

[AW97] B. A. Allan and A. W. Westerberg. The ASCEND IV Language Syntax and Semantics. Technical Report EDRC 97-, Engineering Design Research Center, Carnegie Mellon University, http://www.cs.cmu.edu/ ascend/doc/C/ westerberg/AscendSyntax.ps.Z as of 10/96, 1997. In preparation.

[BCP89] K.E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. North-Holland, 1989.

[Mon96] Jon Ingar Monsen. Department of Mechanical Engineering. Master's thesis, University of Trondheim, Norway, 1996.

[Ost94] J. K. Osterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PMW93] P. Piela, R. McKelvey, and A. Westerberg. An introduction to the ASCEND modeling system: its language and interactive environment. *J. Manage. Inf. Syst.*, 9(3):91–121, Winter 1992-1993.

[RH93] K. Radharkrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver of Ordinary Differential Equations. Technical Report UCRL-ID-113855, Lawrence Livermore National Laboratory, December 1993. NASA Reference Publication 1327.

[Wes96] A. W. Westerberg. ASCEND Modeling Language and Environment Notes. CAPD short course notes, November 1996. http://www.cs.cmu.edu/ ascend/ doc/C/Westerberg/AscendIntro.ps.Z.