# PLANS FOR ASCEND IV: OUR NEXT GENERATION EQUATIONAL-BASED MODELING ENVIRONMENT[1]

Arthur W. Westerberg
Kirk Abbott
Ben Allan
Dept. of Chemical Engineering
and the Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213

*Abstract*

We discuss features of the current version of our ASCEND modeling environment and then many of the modeling insights we have obtained from using it. From this experience and what others are learning when using their environments, we suggest ways we intend to increase the scope and size problem we can solve with a future version of ASCEND.

**Introduction**

The goal of the research group of the first author for the almost three decades has been to improve our ability to develop and solve process models. In the late 1960s, we worked on extending the flowsheet tearing algorithms to the solving and optimizing of models described only by their equations. By the early 1970s [Edie and Westerberg, 1972] we concluded that solution algorithms based on embedded tearing were fatally flawed, in that they frequently introduced singularities in the solving process not present in the original model. Following the lead of Hutchison and his students [Bending and Hutchison, 1973], we switched to using the Newton-based sparse matrix methods electrical engineers were developing to solve electronic circuit models. Both the tearing approach [Westerberg and deBrosse, 1973] and this approach readily supported using generalized reduced gradients for optimization. The switch to using sequential quadratic programming methods -- to remove the last "loop" in the computation -- followed in the late 1970s, and we were able to optimize 5000 equation models having 10 degrees of freedom in our ASCEND II system [Berna et al, 1976, 1980; Locke and Westerberg, 1983]. We knew that these methods had to be more robust, but they promised to be about as efficient as we were going to get, taking only two to three times as long to optimize as it took to solve. At the prodding of Dean Benjamin, a student in the group, we wondered how people were going to "feed" these systems [Westerberg and Benjamin, 1985]; i.e., how can engineers efficiently develop models of that size. Peter Piela joined the group in 1984 with the project to "create a programming system where an engineer could create, debug and solve a new model one order of magnitude more rapidly (in terms of his/her time) than possible with current technology." He took this goal to heart, with the ASCEND III system being the result [Piela, 1989; Piela et al, 1991/92]. It featured two parts: a modeling language based on object-oriented concepts and an interactive user interface for debugging and solving. Our current group is using and improving this system in their research, including the solving of complex processes such as azeotropic distillation columns.

We have learned many things from this work, which we share in this paper. We are also ready to extend this system, i.e., to create ASCEND IV. We discuss here our ideas on the features this system should have, ideas which have come from many sources: the work on Omola by Mattsson and coworkers in Lund [Andersson, 1989], the work of

[1] Presented at AspenWorld '94, Boston, Massachusetts, November, 1994.

Stephanopoulos on MODEL.LA [Stephanopoulos, et al, 1990a,b)], the work of Barton and Pantelides [Barton and Pantelides, 1991; Pantelides and Barton, 1992] on gProms and of course on our own experiences.

**Equational-based modeling vs. modular modeling**

What are the reasons one would want to do equational-based modeling? One argument often made is that modelers should only supply the equations, and the system will supply the solution method. Writing the equations without prescribing how to solve them is much less work, sometimes stated to be something like one sixth as much work.

The real advantage, however, is reuse as the equations can be a separate concept from how to solve them. Once one has created them, one would like to use them in any of a number of ways: e.g., for simulation, optimization, dynamics, etc., or any combination of these. This reason is also emphasized in a recent paper by Pantelides and Britt [1994].

Current modular systems (conventional flowsheeting systems such as Aspen or Pro II) have two significant advantages over most equational-based systems. First, the modeling is generally only by configuration; thus it is fairly easy for the user to get the degrees of freedom right. A user wires together predefined parts where the modeler of the parts has already made the decisions on what someone who plans to use this model has to specify when using it. Second, a model is both the equations and how to solve them; thus a unit operation model such as a distillation column can contain domain knowledge on how to guess initial conditions for it and how to overcome many convergence problems for it.

We suggest that these last two advantages for modular systems must not be lost in modern equational-based modeling. We will suggest here the properties that an equational-based system must have to gain back these advantages. Further we have created the ASCEND system specifically to demonstrate how one can incorporate these properties.

**The ASCEND III system**

The ASCEND system is in its third generation. It is an environment to support equational-based modeling in any technical domain. Most of our models have been in chemical engineering, but others have used it in architecture and in mechanical and electrical engineering. It comprises two main parts: the ASCEND modeling language and the interactive user interface for debugging and solving.

*Object oriented language concepts*

The modeling language in ASCEND is based on object-oriented concepts. It has both a declarative section where one defines the variables and equations that are to be true when one solves such a model and a procedural section to aid in recapturing the properties mentioned above. We list here many of its features.

**Strong typing**: One creates an ASCEND model by establishing a hierarchy of type (concept) definitions from which one constructs it. ASCEND supports strong typing where one must declare every part of a model to be of a previously defined type. This declaration is done using the **IS_A** operator. For example one might declare a variable T IS_A temperature or a more complex part s1 IS_A molar_stream. Type definitions for variables are called atoms which we have used to define such types as temperature, pressure and molar_flow. We can assign attributes to these type definitions for variables such as dimensionality, nominal value, and lower and upper bounds. Thus we can easily construct a model where all temperature have the same nominal value of 300 {K}.

More complex concept definitions, which we call models, can involve parts which themselves are instances of previously defined models, atoms, and which then also include equations that can be in terms of any of the variables in the model (including those within the parts). Model definitions can contain parts that contain parts that contain parts, etc., to any level.

This use of strong typing is useful to detect errors involving model misuse. It permits ASCEND to have principled handling of dimensionality and units, where perhaps a third of our modeling errors used to arise.

**Inheritance**: ASCEND's **REFINES** operator (which is often called IS A in object oriented systems) supports inheritance. An example would be to say that

MODEL liquid_stream REFINES stream;

To refine a previous type definition means that the new type is everything the old one was plus whatever the modeler then wishes to add. The main purpose for this operator is to allow a modeler to state which types this new type can replace in previously defined models. If "liquid_stream" refines "stream", one can use it in place of a stream. Inheritance supports both the merge operator and the deferred binding operator to follow.

**Merging**: One uses the **ARE_THE_SAME** operator to merge two parts contained in a model into a single instance which will then have two names. It is much like an equivalence operator. The ASCEND compiler does three things for a merge: (1) checks that the two parts are type compatible, (2) puts all the variables for each part into common storage locations and (3) lets only one instance create its equations. This last reflects that a model contains both variables and equations. Merging with the ARE_THE_SAME instruction is a very natural way to configure a complex object out of simpler ones. We shall

return to this instruction later when we talk about modeling "without mentioning equations."

**Deferred binding**: In ASCEND one can "reach" inside a part and upgrade the type of one of its parts by using the **IS_REFINED_TO** operator. For example suppose a modeler has included a column called C1 inside her flowsheet model definition. She can alter the type of a liquid_stream inside that column by stating that the liquid_stream is a Wilson_liquid_stream as long as Wilson_liquid_stream is a refinement of liquid_stream. Deferred binding reduces the number of type definitions required by a modeling system. In some systems, if one wants to upgrade the type of a part, one must copy the existing model and edit the copy to show the new type, creating another type definition within the system. Lots of added copies, each a minor variation of the original, can easily result.

Deferred binding is also available through the solving interface. When used, one can pick out a part of a model one has loaded and is solving and ask that it become a more specialized type. The system alters the type and restarts the compiler to propagate the implications of that change throughout the current instance. Values for variables the system previously computed are the initial values for the new more refined instance. Thus one can solve a flash unit using constant relative volatilities, then defer bind the physical property instances to be nonideal types, compile this change into the flash instance just solved and resolve as a nonideal flash model.

**Type propagation**: If our modeler alters one liquid_stream inside a column to be a Wilson_liquid_stream, she probably wants all streams of type liquid_stream to become of type Wilson_liquid_stream. She would not like being required to know of all the liquid streams within the column to accomplish this change. The **ARE_ALIKE** operator permits the propagation of type. The person writing the column model can state that all parts of type liquid_stream within his column model ARE_ALIKE. When our flowsheet modeler makes one liquid_stream within the column C1 into a Wilson_liquid_stream, the system propagates this type to all other liquid streams with which it is alike.

**Universal**: Some parts in a model should be globally defined. For example the constants that describe the physical properties of methanol would be the same in all models. Declaring a type to be **UNIVERSAL** (e.g., UNIVERSAL methanol REFINES component_constants;) makes all instances of it only one instance, as if one had merged them all into that one instance.

**Arrays of anything**: In ASCEND one can ask that any instance be defined over a set either of integers or symbols. Thus one can have a mole fraction defined over a set of component names or a flowsheet defined over a set of time points, the latter being useful when collocating a model containing differential and algebraic equations over time.

**Set manipulation**: ASCEND has a full complement of set manipulation capabilities. For example, it allows one to state that a set is each component name in the set of all component names such that that name is in any of three other sets. Set manipulation is crucial for defining finite element meshes in a natural way or for implementing a Unifac model for physical properties.

**Case statement**: We are about to incorporate the work of Joe Zaher which adds the case statement to the ASCEND language. As Joe is implementing it, ASCEND will allow one to describe a model that operates in different regions such as a flash model that can operate in the subcooled, two-phase and superheated regions depending on the temperature and pressure levels for it at the solution of the problem. In each region a different set of equations holds. The solver has to select the region in which to find the solution as well as solve the equations in that region. Initial guesses may well be in another region, and the solver has to be able to move from it to one where it can find a solution. We do not demand that the same variables are in the equations in the different regions. Joe has had to discover how to do partitioning and precedence ordering, how to aid a user to pick which variables s/he can fix, and how to move across boundaries both when solving and when optimizing such a problem.

**Methods**: One can attach methods to any ASCEND model definition. We put these in originally to allow a modeler to establish initial guesses for the values of all the variables in a model. We subsequently discovered that they were crucial for getting the degrees of freedom right in a complex model built of parts which are themselves built of parts, etc. In constructing our models which we put into our libraries, we require that each model definition contains a method to fix exactly enough variables in it to make it into a set of n equations in n unknowns, i.e., to make itself square. A model comprising several parts can then send a message to each of its parts asking it to fix its variables so it is square. The outer model then makes itself square by making a few alterations to the settings done by its included parts, something that is much easier to do than to start from scratch to get itself square. Our models also contain methods to rescale variables.

When solving a model, one can ask that any part create its equations and send them to the solver. The solver will then solve only the equations for that part. It is essential that the part be square. Sending a message to its "square yourself" method accomplishes just that. If the part did not have this method attached to it, such a solving of parts would be very difficult to accomplish.

If a model definition is a refinement of another, it inherits the methods of the other unless the modeler decides to create a method with the same name, in which case the new definition replaces the earlier one.

**Scripting**: To capture the experience in solving, one needs the ability to attach the steps to solve to a model definition, which ASCEND accomplishes through the use of scripts. A script captures the sequence of commands one has executed when solving it through the interactive interface. It can be replayed at any time for the model. Thus when one has learned to solve a model, he can capture it in a script and pass that learning to others. Since the manipulations possible through the interface are many, scripts can capture very complex solving procedures. For example, one can ask that each of the trays in a column be isolated, made square and solved (a tray-by-tray initialization). The scripts can also capture a sequence of solving, deferred binding to make parts more complex, resolving, etc. Scripts and methods are our means to attach the learning one does to solve a model to the equations that define it, gaining back one of the advantages we described earlier as belonging to the modular approach to modeling.

*The user interface*

The user interface partitions the problem of debugging and solving a model into many different steps: loading and compiling (LIBRARY tool set), maintaining compiled instances (SIMULATIONS tool set), browsing to see what is in a compiled instance and to isolate and examine any part of it (BROWSER tool set), solving and detecting and diagnosing convergence problems (SOLVER tool set), changing the units (e.g., kmol/s or $m^3$/kg) with which to display variables (UNITS tool set), capturing and running scripts (SCRIPT tool set), displaying models and instances of models (DISPLAY tool set) and finally, but not least, collecting parts of the model to be displayed together much like one might copy parts of a spreadsheet into a single page so the parts can be seen on one screen (PROBE tool set). This partitioning is a model of how the user would like to approach solving his/her problem. The approach to developing the interface was to set up a team of developers and users with the team charter being to improve the usability of the ASCEND system. Many hours of observation and discussions among users and developers led to substantial changes and improvements to the interface [Piela, et al, 1991/2]

Each of these tool sets has individual tools to aid the user to solve an instance of a model and, if solving fails, to find out why and to make suggestions as to what changes may overcome this failure. For example, the system can display the incidence matrix for the problem, where each incidence, when moused, tells the user which variable and equation s/he just touched. If many variables and equations are in a block to be solved simultaneously (visually obvious from looking at the incidence matrix), there is a tool called the debugger that allows one to list all the equations and variables in that block.

If the problem is numerically singular, it can be that there is a dependent or inconsistent equation in the model or that one has selected the degrees of freedom in such a way that the remaining problem contains a dependency. Tools in the system aid the user to explore which might be the problem. One tool tells the user that trading one of a given set of fixed variables for one of a given set of computed variables will overcome the numerical singularity. If the system fails to discover these sets, another tool will report which equations appear to be mutually dependent based on a local linear analysis. The user can alter the values for the dependent variables and, if the same equations remain related, become very suspicious that these equations are dependent, even for a nonlinear model.

**What we have learned using ASCEND**

We are continually evolving the ASCEND system. It has been quite similar to its present form for about four years so we have been able to use it for setting up and solving a wide variety of models, ranging from azeotropic columns to dynamic batch column simulation, from process optimization to optimal control problems. In this activity (and previous modeling activities with earlier ASCEND incarnations - like ASCEND II) we have learned a number of useful things about process modeling that we list here. We will list our beliefs and give brief arguments supporting them.

*Modeling is a design problem*

Modeling is a design problem. It is not just coding. We should carefully lay out our goals for the problem at hand. Do we want it to be fast, accurate, or what? Will it require the use of extensive physical property computations? We need to enumerate the alternative decisions we may be willing to make in solving it (which languages, which platforms, which assumptions, etc.) and devise methods to search this space. Most important is that we establish tests to verify we meet our goals for the model. We need to decide if we are going to start from scratch in creating it or if we will be using previously written models as a starting point. It is a good idea to do this type of planning before we set pen to paper or fingers to keyboard to write the code.

*Elegance is necessary to manage complexity and maintainability*

At one extreme there are the so-called "expedient" modelers who say that a model is good enough if it gives the answer. The code can look like spaghetti, and they do not care. At the other extreme are modelers who will never turn over a piece of code because it is not pretty enough. We advocate being close to, but not at, the elegant extreme. An example of elegance is that we should treat all instances of the same type in a model as close to the same way as possible so we can just look at a model to discern it correctness with respect to that concept. All streams should be treated the same. We should not include a statement that the sum of mole fractions add to unity for

some and not others, for example. Elegance often gives a model that is more general than we expected when creating it.

The remaining rules are more specific to coding of flowsheet models.

*In flowsheet modeling, model material balances using molar flows*

Material balances written in terms of molar flows are linear. If one is solving using a Newton-based method where the method often takes full Newton steps, the material balances are satisfied after each such step. Solving thus tends to be along a path satisfying the material balances. A useful study to carry out would be to demonstrate whether this rule is a good one.

*In flowsheet modeling, equilibrium must be stated using mole fractions and not in terms of molar flows*

This rule is more easily justified than the previous one. Examine the following equations which are a way to express vapor/liquid equilibrium.

$$y_i = K_i x_i \quad = \quad \frac{v_i}{V} = K_i \frac{l_i}{L}$$

$$\Rightarrow \quad v_i L = K_i l_i V \quad for \quad i = 1, 2 \cdots n_{comps}$$

In this last form, we can see that these equations are satisfied if V is zero (as then all $v_i$ are also) or if L is zero (then so are all $l_i$) as well as at the point where the first form is satisfied. One has introduced two spurious roots into the problem. The problem is that the ratio of $v_i$ to V becomes zero over zero as V goes to zero, a well defined limit known as $y_i$ which the computer fails to see if one does not use mole fractions in these equations.

We are, therefore, advocating the use both of molar quantities for material balances and mole fractions for equilibrium when defining a stream in flowsheet modeling. We always do both.

*A stream and a holdup should always have its state as a single concept within it. A state contains all the intensive variables and all the equations that we can write among them. A stream is then a state and a total flowrate while a holdup is a state and an amount.*

Intensive variables are those whose values do not change if we change the flowrate of the stream or the amount of a holdup: temperature, pressure, mole fractions, molar enthalpies, molar Gibbs free energies, molar volumes and the like. We might call this collection of variables and all possible independent equations which we can write in terms of them the *state* of the stream or a holdup. Then a stream is a state and a flowrate while a holdup is a state and an amount. If we know $n_{comps}$-1 mole fractions, the temperature, pressure and assume the phase for a stream or a holdup, we can compute all the other intensive variables mentioned above -- without knowing the amount of the holdup or flowrate of the stream.

We use the simple stream splitter to motivate this rule. We specify $n_{outputs}$-1 split fractions which we use to split the flows, stating, for example, that each of the four exit streams gets 20%, 30%, 10% and 100-(20+30+10) = 40% of the feed flow to the splitter. We then also require that all the states for all the streams ARE_THE_SAME, in the sense we described this operator above for the ASCEND modeling language; that is, all the intensive variables defining the states for all the streams are stored in the same storage location and only one of the instances is allowed to generate its equations. Even if the equational-based modeling environment does not support merging using the ARE_THE_SAME operator, this way of thinking is one that allows one to get this model written correctly.

Thinking this way we get two benefits. First, the stream concept for all streams is the same. We do not make a special case of the output streams. Second, no matter how many intensive variables we add to the definition of the stream, this stream splitter model works. The splitter is a set of split equations and the merging of all the states. A corollary of this rule is that one should always test the stream concept by using it with an existing stream splitter. Almost all attempts at this test, until one understands the issue above, will lead to a model that generates too many equations.

*Understanding degrees of freedom for chemical process requires a profound appreciation of the Duhem's other theorem which states*

> Whatever the number of phases, of components or of chemical reactions, the equilibrium state of a closed system, for which we know the total initial masses of each component that will ultimately appear in it, is completely determined by two independent variables."

First this theorem is not the classical phase rule as it talks about the extensive properties of a stream while the phase rule talks only about the state as we defined it just above. From this theorem we argue that, if one specifies the molar or mass amounts for all the species in an equilibrium stream and the temperature (alternatively, enthalpy per mole) and pressure for that stream, we may then derive all other thermodynamic properties for it. Each stream introduces exactly $n_{comps}$+2 new variables to a model no matter how many variables we use to characterize it. Added variables always bring along an equal number of equations to define them. Any model that violates this rule cannot be right.

*An equational-based approach must provide the modeler with the means to aid a subsequent user to get the degrees of freedom right for his/her model*

We made the argument for this statement when we described methods above for ASCEND. Essentially each type should have a message to "square itself" so a modeler of a complex type can write the "square itself" method as the difference of what the parts do and what this model requires, a step that is markedly easier than getting it right from scratch.

*A modeling environment must aid a user to make proper trades when changing degrees of freedom in a model*

Without these types of aids normal human beings cannot be expected to adopt the equational-based approach to modeling. An example is for the system to drop a window and tell the user that his/her system is not square and that he/she should fix one of the following variables if underspecified or release one of them if overspecified. In ASCEND, one can ask the system to show only those options that are within a specific part of the instance being solved. For example, one could ask the system to list variables to fix that are in the column -- if there are any.

*The system must allow the modeler to do his/her own scaling of equations*

Scaling is not understood well enough at this time to exclude the modeler from participating in its definition. There are times when only the modeler knows how a variable is to be scaled. The system must allow a user to create a method which, when called, will do the scaling s/he desires for that model. Again this step is much easier if it can be done by calling scaling methods for each of the parts and fixing up the differences.

Scaling can affect the convergence of a model in two ways. Poorly scaled equations will make it difficult to detect convergence when it may in fact have occurred. Poor scaling of variables and equations will affect the pivots a sparse matrix package will use when solving the Newton equations. The steps taken by a Newton method are scale invariant if one computes them using infinite accuracy. Unfortunately computers have a finite word length, thus one will find that a poor pivot selection can introduce significant numerical errors in solving a set of linear equations. Try solving the two equations

$$\begin{bmatrix} 1 & 10^{-6} \\ 10^{-6} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

using first the diagonal pivots and then the off diagonals as pivots where you can retain only four significant digits in any computation. In the latter case the solution is not only incorrect, it is independent of the 2 in the right hand side of the second equation.

*Methods are absolutely necessary for modeling in an object-oriented setting*

This observations summarizes a number of the above observations.

*Scripts or their equivalent are necessary for a modeling environment*

The environment must allow a modeler to creep up on the solution in almost any way desired so the modeler can gain experience in solving the model and can then encode this experience in the final model description. Scripts are a mechanism to encode and pass this experience to others.

*Avoid the divide operator in both the model equations and in the forming of the partial derivatives needed by the Newton method if the divisor has any chance of approaching zero*

With rare exceptions in our models, we do not allow the divide operator. In fact, it might be a good idea to prohibit its use. We would except there are some modelers who would never forgive us if we did. The divide operator invites divide faults as variables get near to zero while trying to solve. We do not allow divide in the evaluation of Jacobian elements either. Thus we advise that an equation of the form

$$y + \ln(x) = z - 3$$

should be replaced by the two equivalent equations

$$\exp(v) = x$$
$$y + v = z - 3$$

In the former form, the partial of the equation with respect to x leads to the Jacobian element 1/x while the latter form avoids division by x. The latter form has a much larger region from which initial guesses will converge.

One typically has to add equations and variables to the problem to rid the model of divides, but it is worth it.

*Watch out for completely recycling species in steady-state modeling*

Think of a refrigeration cycle. Typically the refrigerant completely recycles in such a process. It appears to the network analyzer in a sequential modular flowsheeting system that the flow is a tear variable to be guessed and converged when in fact it is a variable whose value the modeler can fix. Not realizing it is a degree of freedom for the problem is a mistake. This same problem shows up in equational-based modeling as the inclusion of a redundant material balance equation. The issue here is that one can introduce singularities into one's models by configuring properly written models which do not contain any

singularities. The process of hooking up the parts introduces the singularity.

**Equationless modeling**

Several studies have investigated allowing a modeler to state the assumptions behind the creation of a process model and then have the system write the material and energy balances and so forth for the resulting model. We would like to argue that we can emulate such an approach within the ASCEND system by developing the proper hierarchy of concepts. Modeling of the kind suggested for equationless modeling is then limited to configuring of instances of these concepts using the IS_A, IS_REFINED_TO and ARE_THE_SAME operators.

One approach is to invent the concepts of a region and a transfer mechanism. Roger Sargent at Imperial College and, independently, Jack Ponton at Edinburgh proposed this idea to the first author while he was on sabbatical in Edinburgh in the fall of 1992. A region is a quantity of material which one wishes to treat as being homogeneous. It is essentially what we have called a holdup above: a state and an amount. In addition it has an arbitrary number each of input and output ports. A transfer mechanism connects an output port of one region to an input port of another region with positive flow being designated as going from output to input. The mechanism transfers energy and/or material with the amount depending on the properties of the two regions it is joining and the parameters that characterize the transfer mechanism.

In ASCEND we model a region as a holdup with a state. It has four arrays, one each of input material ports, input energy ports, output material ports and output energy ports. Dynamic material and energy equations complete this definition. The summations to write these balance equations are over all elements in the appropriate sets defining the inputs and outputs. We do not declare the members of the sets within this definition. We buried the equations in the region definition. They need not be visible to the person using the region as a predefined concept.

We model the transfer mechanism as having an input port and an output port. We include also the equation that relates the transfer rate(s) to the states of the regions being joined and the mechanism parameters. For example the mechanism could be a valve, and the model includes the equation to relate flow through the valve to upstream and downstream pressures as well as to the parameters describing the valve geometry.

An example of a complex model we can construct from such building blocks is that of a dynamic model for a flash unit based on mass transfer mechanisms between the vapor and liquid phases. One region is the liquid region; the second is the vapor. We model the feed, any heat input, the vapor and liquid products streams and the mass transfer between the vapor and liquid regions as instances of appropriate transfer mechanisms. It is in this outer model that includes the regions as parts that we declare the members of the sets over which we define the ports for each of the regions. For example, we reach inside the instance defining the liquid region and say that the set listing its input material flow ports contains the single symbol 'feed'. We declare the elements 'liq_product' and 'vaporization' to be in the set listing the output ports. Note that, at this level of modeling, we are only configuring from predefined concepts. We are not writing equations. It is in this sense that we suggest that the operators in the ASCEND language are sufficient to emulate equationless modeling in a very useful way.

An interesting question is whether we should partition a region into phases which are in equilibrium with each other or whether equilibrium should be treated as a special transfer mechanism. We based our thermodynamics library on the former; i.e., we partition a region into equilibrium phases. We associate the output ports with each phase and the input ports with the region as a whole.

Transfer mechanisms and regions are a very elegant way to model chemical processes. They are very domain specific. We still have to program other concepts which introduce equations (e.g., how do we write the thermodynamics library itself); regions and transfer mechanisms are not sufficient for all our modeling needs.

**The next generation ASCEND**

When we first had the essential features of the current ASCEND system available, it was difficult for us to see how to improve it. Experience has shown us several ways we can extend it to create a next generation of ASCEND. Succinctly, we want to solve much larger models and we want to increase the scope of things we can model.

*Solving much larger models*

We routinely solve models of the size of 5000 to 10,000 equations and can solve upwards of 25,000 algebraic equations at this time. The actual size of course depends on the structure of the model. If the model is one large block of simultaneous equations, the solving time and space taken is much larger than if the model fully partitions into a sequence of single equations which the solver solves one at a time. The number of equations is, therefore, often very misleading when one is stating what can be handled by a simulator. Maximum block size is probably a better way to describe the size problem one can solve.

We want to solve models involving half a million equations with block sizes that can number in the tens of thousands. The question is how? We need to solve faster and in less space. Also, we need methods to verify we are solving the problem we intend. Finally we need methods to state and debug such large models. The approach we advocate is again to build large things from smaller things.

**Procedures**: One way to increase the size of models that we can solve is to use the lesson one has learned from sequential modular simulation; namely, encode parts of the

model as well debugged code that we know solves robustly. This means using procedures. These can have two uses in the larger model: to supply equations and to supply initial guesses. We must accomodate both, the former in the "declarative" part of the model definition and the latter in the methods attached to model definitions. We have already implemented the use of procedures, and one can use them in both these ways. Further the implementation uses dynamic loading so they need be in memory only while they are executing. That reduces space requirements substantially.

Why are procedures useful? First of all they eliminate internal variables, reducing the size of the outer problem. Thus they can reduce the space required to solve, superficially appearing to trade reduced space for increased time. Second, there may be a highly specialized and efficient way to solve the equations in the unit model (a distillation column generates a block tridiagonal structure for its equations which one can solve efficiently [Napthali and Sandholm, 1971]). Thus the overall model may actually be faster in solving. Third, if tearing is used to solve the model within the procedure, one could map its tear variables and functions to the outside to become a part of the outer problem and not converge them internally, an idea that Bill Johns showed could be extremely effective in the early 1970s while he was at ICI.

Finally, the procedure could create its full set of residuals and partial derivatives during a Newton step on which it would carry out a local forward elimination, passing a reduced set of linearized equations to the outer problem. When the outer problem back substitutes it equations, it triggers the inner procedure to back substitute its also. This idea is really a disguised multifront method [Westerberg and Berna, 1978]. Here the advantage is keeping the pivoting localized which may reduce fill substantially over the fill a general purpose sparse linear equation solving package creates.

The following analysis shows the potential for speeding up the solution process for a multifront method. Suppose that the solving time for a model grows quadratically and that the quadratic term dominates when problems are 5000 or more equations in size; i.e., assume for larger problems that solving time is equal to $aN^2$ for accomplishing the L\U factorization when solving using a Newton based method. Suppose we can break our problem naturally into M roughly equally sized parts. Then the solving time for the parts will be $Ma(N/M)^2$ which equals $aN^2/M$ rather than $aN^2$. We reduce the linear solving time by about 1/M. For M about 10, we are looking at the potential for an order of magnitude reduction in the solving time for the linear equations of the inner Newton step.

The speedup comes because the solving package can control sparsity better; it keeps the pivoting localized in the parts, keeping at bay the time when the sparse matrices become full matrices during the elimination process.

**Recursive ASCEND**: We can next wonder how we are to produce those procedures if they do not yet exist. One way is to use ASCEND recursively. We see two ways to do that.

First we can work initially and independently on a model type definition, learning to solve instances of it for many different example problems. We can pick the variables that we find best to specify, leaving a computation for the remaining that proves robust. For example, we typically find that we can solve simulations much easier than we can solve design computations. We propose then that we push a button and have ASCEND write C code that can solve the model in this final form. The C code uses a fixed way to solve and is very fast as it does not have to trace through the pointers our general purpose solvers have to when tied to the current very flexible ASCEND data structures. In other words, if we remove the need for flexibility, we can remove the need for complex data structures. The modeler must also define the interface to this code to tell the user which variables it expects as inputs and which it will give back after solving the model. It will also evaluate and pass back needed partial derivatives of its computed variables with respect to its input variables.

The C code can create a reduced set of residuals and partials or it can process the sub model as in a multifront method for embedding it within the larger problem.

A second approach to use ASCEND recursively, which is useful when debugging, is for ASCEND to reopen itself in another set of windows on this part of the problem when the it invokes a procedure. The user can then work interactively on the part to get it to solve. When successful this part can send back its residuals and partial derivatives for the reduced problem. The outer problem will be much smaller. The major difference is that instead of creating a C code to solve the inner problem without aid from the user, the user aids in solving the inner problem.

**Embedded complex parts**: The parts that one embeds may be operate in a manner that is more complex than just supplying equations. For example, the part may itself solve a simulation or optimization problem subject to inequality constraints. It may operate either as a glass box, a gray box or a black box. The outer problem can see everything that is inside an included glass box while it can see nothing inside a black box.

What if the inner problem, operating as a black box, fails. Should the outer problem fail or should it attempt to work around failure of the inner box? Kirk Abbott (second author) is looking at the coordination problem for these classes of problems. He has looked at the inner problem being a black box simulation that can report back either a successful solution for its outputs given its inputs or a message that it has failed because the inputs lead to an infeasible solution. He has also looked at the case where a glass box inner problem contributes a part of the overall objective function for an optimization problem. Methods to coordinate this problem type include two-level

optimization methods as well as methods based on Benders decomposition.

The really hard problem to coordinate is one where the inner problem is optimizing an objective that is unrelated to the objective function of the outer problem, leading to a multilevel optimization problem [Clark, 1983]. An example would be for the inner problem to find a solution to a collocation problem where grid point locations are the result of minimizing some modeling error criterion while the outer problem is optimizing a cost objective. Another is the classic problem of two competitive companies each optimizing its own objective while being aware of the other's objectives.

We would like the ASCEND system to allow a user to set up and solve these kinds of coordination problems, with numerous aids to diagnose modeling errors and to suggest ways to overcome them.

*Increasing scope*

The second way to extend ASCEND is to increase the scope of the problems one is able to set up and solve using it. At the present time, it can solve and optimize problems where the model contains only variables which can be represented as reals. While that allows us to model many classes of problems, we would like to broaden the scope significantly.

**Integers, binaries, logicals, etc.**: The first way to broaden the scope is to allow a modeler the ability to use more types of variables: complex, integer, binary, logical, stochastic, interval variables and field variables. Interval variables are expressed by stating an upper and lower limit for each. They obey the algebra of intervals in any equations in which they appear. The temperature variable in a slab is a field variable. One can define it implicitly by writing a partial differential equation that must hold for it over a field. It has values everywhere in the slab for all the times in the specified time period of the simulation.

We can add integer and binary variables with absolutely no change to the current ASCEND compiler by noting that the type integer is a refinement of real and binary is a refinement of integer. For each real and any refinement of a real, the system sets aside a double precision real value slot to hold its value. A solver can then count on having a real value as a relaxation of an integer if that solver chooses to solve first by removing the integer constraint. The effort to add integer and binary variables is to add the solvers that can deal with them, such as an MILP or a MINLP solver.

We already use Boolean variables as flags, but they cannot appear as unknowns in a model. We have to alter the compiler to allow logical equations to be a part of a model. We intend to maintain the equational-based approach when including them; namely, we want modelers to state the logical equations that must be true at the solution to the problem but not how to solve these equations. Thus we will allow an equation of the form

```
T1>300{K} = P<10 {atm};
```

This equations will have the equivalent of a residual attached to it. This residual will indicate whether the equation is true or not. If not, it will also indicate why not. We envision feasible path solvers that will insist on such equations being true as they iterate to a solution as well as infeasible path solvers which will not.

We have not thought our way through allowing complex, stochastic and interval variables as of yet; therefore, we shall not comment on them here.

Variables that have values over a field are more interesting. In a personal communication while the first author was visiting Imperal College in 1992, Pantelides describe adding such variables to gProms. As our group has reflected on them in the ASCEND environment, we see them as a simple extension of the ASCEND modeling language. Namely, they are variables defined over an infinite set. ASCEND already supports allowing one to define anything over a finite set. Thus the linguistic extension is pretty straight forward. However, the compiling and solving implications are anything but straight forward. For finite sets, we require the sets to be explicitly defined before compiling commences. The compiled data structure contains storage space for every variable and equation defined over such a set. The intervals over which one intends to define an infinite set can be defined, yet the compiler cannot set aside the space as the solver controls how many points it will need in time and space as it solves. For example, it will chose the number of steps it needs to solve a set of differential equations while marching forward in time for a dynamic simulation. What can be compiled, however, is one instance in time and space of the model as we already do to solve a dynamic simulation. In principle, the solver uses the compiled instance to determine the time derivatives and algebraic variables, given values for the states. The solver uses this same model repeatedly for different values of the states to integrate forward in time.

The expressiveness of the modeling language increases enormously with the extension to allow field variables. Every variable can be thought to be available at each instant in time and at each point in space over which one defines it. We have used the ASCEND constructs that exist or that we have discussed in this paper to model on paper both the description of a batch process and the tasks that one wishes to perform using the equipment in that process. We have even modeled irreversible transitions. In other words, the language is rich enough by itself to model both processes and tasks.

**Calculus-based modeling**: Ben Allan (third author) is currently looking at how to add calculus operators to the ASCEND modeling language. Underneath the system will do all the manipulations to set up such models by using exact numerical differentiation methods [Ponton, 1982] rather than symbolic manipulations. Work in the last five

years at Argonne National Laboratory extends these ideas and is producing code to implement them. Allan's first goal is to provide tools to aid a modeler overcome an index problem if s/he creates one in a dynamic simulation model. This goal motivated the extensions we shall mention in a moment.

An index problem occurs whenever the model places an algebraic constraint among the state variables in the simulation. An example is for the modeler to ask for a dynamic flash model to run at fixed pressure when s/he has modeled the vapor phase as having holdup. Any good control engineer knows that such a specification is not possible. Rather one must place a pressure control loop on the flash model to hold pressure near to a constant setpoint. One can select the pressure to be a state variable for such a model; holding it constant is to place an algebraic constraint on it. Another example is to model two tanks coupled with a pipe which allows flow in either direction. If the resistance of that pipe becomes very low, the two heights are algebraically coupled; i.e., they should become equal to each other.

An index problem results because the model does not explicitly contain all the constraints it can, complicating life for the solver. If a state variable is algebraically constrained, so is its time derivative. For example, saying that pressure is to be constant also says that P'=0, and saying that the two heights are equal, i.e., $h_1=h_2$, also implies that their time derivatives are equal, i.e., $h_1'-h_2'$. The model does not include these latter equations. Solving while unaware of these latter constraints leads to problems with error control and/or to initial conditions that are inconsistent. If we include these equations, we have a well-posed problem with one less state variable, and any dynamic simulator should be able to solve it without any unusual precautions.

We believe that one should be able to request a solution where we hold pressure constant - albeit, with a warning from the system - as there is a solution to this problem. Not everyone would agree with us on this point, however. Some argue that, if the solution is not physical, it should not be allowed. We argue that, if the solution is there to be had, we should try to find it. In requiring constant pressure, one may wish to compute the changes one has to make to a variable that a controller would manipulate. We might wish to see if these manipulations are possible for the controller (they may ask a valve to more than close, for example).

To adopt our approach, we have to discover the algebraic constraint on the state variable and add the time derivative for it to the model. Discovery involves finding singularities in the model. Pantelides, in his PhD thesis, provides algorithms to do this if the singularity is one caused by the structure of the problem, as in the pressure example above. However, it cannot detect the problem if the resistance goes towards zero as the solution progresses, as in the second example above. Here one has to discover singularity by detecting ill-conditioning. We have proposed an algorithm and are working on improving it for problems where the system would find ill-conditioning difficult to detect.

To set up the problem we need to add the time derivative of one or more equations that are in the model. If we allow the system to add the time derivatives of equations to a model, why not let the modeler do it also if s/he chooses to do so. We are extending the syntax of the language, allowing a modeler to say something like (we are still exploring the exact syntax):

```
x$_defn:  x$ = x+y;
x$$_defn: full_der(x$_defn,time)
```

The system will add this equation and introduce new variables x$$ and y$, if they are not already in the problem, where x$ is the full time derivative of x with respect to time. Note the modeler does not state the actual equation. The system will generate it (by generating its Newton equations directly using exact numerical differentiation).

In a similar fashion, we are adding operators that can add the partial derivatives of any expression to the model. Thus one will be able to model the Newton equations directly to solve a set of equations. We will also allow the user to add partial differential equations. An example (syntax not yet fixed) is:

```
T$=partial_der(T,$)+ux*partial_der(T,x);
```

$$(\Leftrightarrow \frac{dT}{dt} = \frac{\partial T}{\partial t} + u_x \frac{\partial T}{\partial x};)$$

With the availability of partial derivatives, we see modelers asking ASCEND to do Taylor Series expansions of functions and with these examining new computational algorithms.

There are interesting implications on what the system can compile and what it must insist one does only to an instance. For example, one cannot ask the present system to create a set containing all the variables in a part. Sets, once specified, are fixed in ASCEND as they determine the size of the data structure resulting from the compile step. A part, through deferred binding, could become more complex and have more variables. The system would have to redefine the set listing all variables in that part, something it cannot do at present.

**In summary**

We have discussed the current ASCEND modeling environment. Using it we have developed many insights into modeling which we discussed here. Finally we present several ways we can improve ASCEND to solve larger problems and to increase significantly the kinds of models we can write.

## References


Andersson, M., Omola - An Object-oriented Language for Model Representation, Lic.Tech thesis TFRT-3208, Dept. Automatic Control, Lund Inst. Tech., Lund, Sweden (1990).

Barton, P., and C. Pantelides, "The Modelling and Simulation of Combined Discrete/Continuous Processes," in Proc. Process System Engng Conf. (PSE'91), Montebello, Ontario, Canada (1991).

Bending, M.J., and H.P. Hutchison, "The Calculation of Steady-state Incompressible Flow in Large Networks of Pipes," Chem. Eng. Sci., 28, 1857-1864 (1973).

Berna, T.J., M.H. Locke, and A.W. Westerberg, "Optimization of Large Chemical Process Systems," 2nd Intn'l Symp. Large Engineering Systems, May 15-16 (1978).

Berna, T.J., M.H. Locke, and A.W. Westerberg, "A New Approach to Optimization of Chemical Processes," AIChE J, 26(1), 37-44 (1980).

Clark, P.A., Embedded Optimization Problems in Chemical Process Design, PhD Thesis, Dept. Chem. Engng, Carnegie Mellon Univ., Pittsburgh, PA 15213 (1983).

Edie, F.C., and A.W. Westerberg, "A Potpourri of Convergence and Tearing," in Chemical Engng Computing, Vol.1, Proc. of an AIChE Workshop, AIChE, New York, 35-39 (1972).

Locke, M.H., and A.W. Westerberg, "The ASCEND-II System - A Flowsheeting Application of a Successive Quadratic Programming Methodology," Comput. Chem. Engng, 7(5), 615-630 (1983).

Napthali, L.M., and D.P. Sandholm, "Multicomponent Separation Calculations by Linearizations," AIChE J, 17(1), 148-153 (1971).

Pantelides, C.C., H.I. Britt, "Multipurpose Process Modeling Environments," Foundations Computer Aided Process Design Conf. (FOCAPD'94), Snowmass (1994).

Pantelides, C., and P. Barton, "Equation-oriented Dynamic Simulation. Current Status and Future Perspectives," In Proc. European Symp. Comp. Aided Engng-2 (1992).

Piela, P.C., ASCEND: An Object-oriented Computer Environment for Modeling and Analysis, PhD Thesis, Dept. of Chem. Engng, Carnegie Mellon Univ., Pittsburgh, PA 15213 (1989).

Piela, P.C., R. McKelvey, and A. Westerberg, "An Introduction to the ASCEND Modeling System: Its Language and Interactive Environment," J. Management Info. Systems, 9(3), 91-121, Winter (1991/1992).

Ponton, J.W., "The Numerical Evaluation of Analytical Derivatives," Comp. Chem. Engng., 6(4), 331-333 (1982).

Stephanopoulos, G., G. Henning, and H. Leone, "MODEL.LA. A Modeling Language for Process Engineering - I. The Formal Framework," Comput. Chem. Engng, 14(8), 847-869 (1990b).

Stephanopoulos, G., G. Henning, and H. Leone, "MODEL.LA. A Modeling Language for Process Engineering - II. Multifaceted Modeling of Process Systems," Comput. Chem. Engng, 14(8), 813-846 (1990a).

Westerberg, A.W., and D.R. Benjamin, "Thoughts on a Future Equation-oriented Flowsheeting System," Comput. Chem. Engng, 9(5), 517-526 (1985).

Westerberg, A.W., and T.J. Berna, "Decomposition of Very Large-scale Newton-Raphson Based Flowsheeting Problems," Comput. Chem. Engng, 2(1), 61-63 (1978).

Westerberg, A.W., and C.J. deBrosse, "An Optimization Algorithm for Structured Design Systems," AIChE J, 19, 355 (1973).