NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making
of photocopies or other reproductions of copyrighted material. Any copying of this
document without permission of its author may be prohibited by law.

INTERACTIVE PROGRAMS FOR PROCESS DESIGN

by

Dean R. Benjamin, Michael H. Locke
and Arthur W. Westerberg

DRC-06-28-81

August 1981 C^

INTERACTIVE PROGRAMS FOR PROCESS DESIGN

Paper to be presented at the Detroit Meeting of the AIChE, August 1981

Dean R. Benjamin
Michael H. Locke
Arthur W. Westerberg

Department of Chemical Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This work was funded by NSF Grants MCS-7822328 and CPE-7801809, and by a grant from the Albert P. Sloan Foundation

Table of Contents

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA

LIST OF FIGURES

Abstract

The paper discusses criteria by which one can assess how friendly an interactive computer program is. Interaction is affected by the size of a typical input/output transaction and turnaround and response times. It is also impacted significantly by the language created for the user/program communications. We discuss where serious problems are frequently created by not attacking all of these aspects carefully.

Two programs illustrate different interactive features. The first, ASCEND-!, is a "desk calculator" for solving physical property and single unit operation calculations. It has several levels of help supported by friendly on-line documentation. It also has an extraordinarily clean structure from the user's viewpoint, allowing him to guess how to use the system. It also attempts to keep the user's view of the **program** entirely in "chemical engineering" terms. The second, ASC£ND-II, is a fully interactive flowsheeting system. It supports the evolutionary development of models **for** complex processes. The paper describes the various interactive features of both systems.

Introduction

Every author of a program for computer-aided design tries to make the mechanical aspects of running the program as unobtrusive as possible. The purpose of such a program is, after all, to free the user's attention from the details of calculation and to focus it on the details of design, not to replace tedious calculations with tedious communications. To this end, the programmer invents a *front end.* or *interface* — composed of elements such as a "problem-oriented" language, pleasing formats for output, assignment of default values for minor parameters, and a manual to describe all to the user. On the merits of its interface, the program is confidently declared to be "easy to use, easy to learn." It is embarrassing to admit, but our experience with some programs we have written is surely familiar: users raise questions whose answers seem obvious. Our reflexive response is, "Look, it's (er. buried) right here in the manual;" but the reflective response is a recognition that the user's confusion is **a** consequence not merely of flawed documentation but also of a poorly designed interface.

The paper is divided into three sections. Section 1 identifies some motives for **the** use of interactive computing in engineering design. We discuss characteristics of the user interface that we feel are essential to achieve the potential of interactive computing. In sections 2 and 3 we describe two programs under development at Carnegie-Mellon University that incorporate some of these principles. ASCEND-I is a simple "desk calculator". Its limited functional capabilities left us free to emphasize the design of its interactive user interface. ASCEND-II is an advanced, equation-solving, flowsheeting system whose clean algorithm and data structures promise an interactive program particularly suited to the requirements of engineering design.

The reader should bear in mind that we speak with two biases:

1. We are academic chemical engineers. Most of the CAD programs we have **used are** programs written for analysis of chemical plants.

2. We believe that the analysis step for engineering design involves essentially two activities: the collection of equations representing process.

and the solution of the resulting systems of equations.

1. Interactive Computing and Engineering Design

Everyone agrees that "interactive computing" is a good thing, but the term has acquired so many connotations that, though it grabs attention, it conveys little more than mood. The term conjures different images for different listeners. So let us take a moment to examine some reasons why the word *interactive* so arouses the libido of the engineering community.

1.1. Measures of Interaction

The words *interactive* and *batch* are widely understood to denote contrasting styles of communication with programs. Below, we define three senses in which programs are said to be interactive. Each is a subjective scale against which people measure the "interactiveness" of a program. We wish to focus on how each measure affects the use of programs for engineering design. Though these observations are neither novel nor profound, they serve to clarify discussion.

1.1.1. Transaction

We will call a *transaction* a cycle of input - computation - output. By *output* we mean any response of the system to the user's input, whether the results of numerical calculation or merely an acknowledgement that the input is free of errors. The use of a program to perform a task involves one or more transactions.

In the view of the user, the "size" of a transaction depends on the complexity of input, the elapsed time from submission of input to delivery of output, and the amount of output displayed. The size of transactions is perhaps the most common distinction between batch and interactive programs: *The user communicates with interactive programs by means of small transactions.* Why is this aspect of interactive computing so attractive to engineers?

Engineering design shares with all creative activities an evolutionary quality: because problems are rarely well-defined, because problems are redefined in the light of partial solutions, because costs must be balanced against benefits when neither are easily quantified, the engineer must "feel the way" toward solutions. Engineering is as much *learning* as it is *doing.* So engineers regard skeptically methods that require omniscient statements of problems. Designs do not spring fully developed from the brows of engineers, but evolve in small steps toward a satisfactory, not necessarily optimal, solution. Herbert Simon has coined the word *satisficing* to describe the process. The activity is properly expressed in small transactions.

Most programs in commercial use for the design of chemical processes, of which FLOWTRAN [Seader74] is the archetypical example, communicate in very large transactions indeed: the input is a description of a flowsheet expressed in several lines of a special language, simulations run for minutes, and one measures in kilograms the pages of tabulated results. We will not dwell on the many reasons for the development of these programs into their present forms, but we have observed in unanticipated consequence of their use: *Large transactions encourage attempts at large problems.* The user who spends hours preparing voluminous input (submitting it a few times to remove syntactical errors) is unlikely to "waste" time investigating small problems. Having at great pains established one solution, the user hesitates to rip out a small section of the flowsheet for closer scrutiny; rather, one tends to make small changes in the input and repeat the large problem, and then to examine only the relevant sections of the output. An understandable reaction, even if poor engineering practice.

1.1.2. Turnaround

Large transactions often grow yet larger in the perception of the user. Since computation and output require no participation by the user after the input has been prepared, these steps of large transactions can be performed at "convenient" times.[1] So batch programs spend lots of time waiting to execute; in time-sharing systems, number crunchers compete with other jobs for CPU cycles; large, outputs are printed last. We'll define *turnaround* as the extra time spent in these various queues.

We can now identify another connotation of interactive computing: *Interactive programs have no turnaround.* The effect of turnaround on the use of CAD progtams is, again, to encourage attempts at large problems. For example, during investigation by case study, one's train of thought should not be derailed. The user will not wait for the results of one simulation in order to plan the next; instead he will blindly call for many simulations in the hope that an useful pattern will emerge.

1.1.3. Response time

Response time is the time necessary to compute the results of a transaction; *i.e..* the time by which a transaction is measured is the sum of turnaround and response time. The programmer has little control over turnaround, which is usually determined by the operating policy for the host computer, but can reduce response time by improving algorithms and data structures.

The programmer must compromise if the speed of an algorithm intended for interactive use cannot be much improved. The processing of a single large transaction can be distributed over several smaller ones. Another compromise is to trade rigor for speed; to substitute a simpler, less accurate, fast technique for a rigorous, computationally expensive one. In section 1.2 we discuss the design of interactive systems, but we mention this here because it is yet another sense in which *interactive* is understood: *Interactive systems have short response times.* Techniques that require long times for computation simply do not lend themselves to an interactive implementation. But by our previous arguments, such techniques have a correspondingly limited usefulness for engineering design.

1.1.4. Interactive Programs *vs* Interactive Environments

The proliferation of interactive operating systems has, within the last decade, changed the mechanics as well as the strategies of programming. Source code is no longer transcribed on punched cards or other intermedium; rather, the programmer enters it directly with a text editor. Engineering programs operating in such interactive environments accrue several of the above characteristics and advantages of interactive computing. Text editors can be used to prepare the input to a simulator such as FLOWTRAN. Most of the turnaround is eliminated if the simulator

[1]At C-MU. convenient means "after 4:30 PW.

can be invoked from the user's terminal, and the apparent size of the output is reduced when read with a text editor.

Many programs for engineering analysis demand several input parameters but little computing power. A modest programming effort can exploit the inherently short response times of these programs: preface each READ statement with a short prompt ("Enter the temperature and pressure of the feed: ") and *Voilaf* Instant Interaction.

Another option is available to engineers skilled in programming and numerical techniques: programs themselves can be quickly modified, compiled, and executed in an interactive environment. In this case, we can regard the statements of the programming language as the "input" of a transaction; utilities of the computer system — editor, compiler, debugger — perform the computations.

1.2. Potential of Interaction

Although the minor changes just described will surely make a given program easier to use and render it more "interactive", such programs hardly realize the potential of interactive computing for assistance in engineering design. The user is merely operating batch programs in an interactive environment.

What then, is missing? How can we build, and by what criteria can we evaluate, programs for interactive design? We suggest some guidelines for the design of CAD tools.

1.2.1. The Grammar of the User Interface

A *grammar* is a set of rules by which the symbols of a language may be arranged in sequences to form sentences in that language. It may contain rules for constructing symbols from other symbols; symbols that cannot be formed from other symbols are called *terminal symbols.* If the words in the *Oxford English Dictionary* are taken as terminal symbols, then the English grammar provides rules for combining words into phrases, phrases into clauses, and clauses into sentences.

Moran [Moran78] extends this familiar notion of a grammar to represent the user interface — "that part of a system that the user comes in contact with physically, perceptually, or conceptually." His Command Language Grammar (CLG) interprets diverse features of the interface — tasks, keystrokes, commands, hardware — as grammatical symbols. For our purposes, the CLG provides a rich vocabulary for discussing the design of user interfaces.

The formalism of Moran's CLG stratifies symbols and rules into levels which we will briefly describe below. The terminal symbols of one level are "inputs" to the next lower level, where they are in turn decomposed. To return to our example of the English language: words, regarded as terminal symbols at a level that deals with phrases, clauses, and sentences, could be regarded as sequences of letters at a lower **level;** a higher level, say, a "book", might take sentences as its terminal symbols. **The** demarcation of levels is somewhat arbitrary, but is most convenient when the **rules** of one level are not expressed in terms of symbols at other levels.

Task Level

The purpose of a program is to help the user to perform certain tasks. The purpose of the Task Level is to identify those tasks and to show how to apply the program to accomplish the tasks. It shows where the program fits in the scheme of things; it answers the user's question, "How do I use the program to do my task?" by showing how primitive tasks (terminal symbols of the Task Level) are combined to perform a given task.

The Task Level also identifies tasks that the user is expected to do without the aid of the program. An engineer may, for example, be faced with the task "design an ethanol plant." Among the subtasks of this ill-defined task, the engineer might list "determine possible feed stocks", "choose a reaction", "design the reactor", "synthesize ttie flowsheet", "simulate the flowsheet". The task "design the reactor" might involve "find a kinetic model" and "solve the kinetic model for varying conditions of the feed." Further decomposition could connect the tasks "simulate the flowsheet" and "solve the kinetic model" with services provided by a simulation such as FLOWTRAN. The engineer must use other means to accomplish the remaining tasks.

The rationale for representing the interface as a grammar now becomes clean the user must translate ("compile") a task into subtasks, successively refining each step until, at the lowest level, he strikes keys on the terminal. The user communicates, at any level, by constructing "sentences" of the symbols at that level. At the Task Level; the symbols are tasks, and sentences are sequences of tasks.

The structure of the Task Level in all but the simplest programs is too complex to elaborate fully. For how many "tasks" is a system that solves linear programming problems written? Users of CAD tools are usually highly trained; in the vocabulary of CLG we would say that the user is expert in the grammar of the Task Level. To be useful for the design of an interface, the Task Level cannot contain the most high-level tasks; the challenge to the designer of the system is to identify those tasks which are most frequently performed (what are the immediate neeas of the user), and conform the system to those tasks.

Semantic Level

If a computer can help perform a task, eventually that task must be expressed in terms comprehensible to the computer. The overriding motive for writing a computer program is to simplify this chore; in other words, the purpose of the program is to provide a language whose symbols more closely match those of the Task Level. It is obvi.ous that "compute the dot product" is an inappropriate language in which to express "simulate an ethanol plant," just as it is impossible to discuss Shakespeare in terms of letter forms; it is equally obvious that "simulate an ethanol plant" is a useful statement only for the manager of an engineering department.

The design of the Semantic Level is the search for this common language. The Semantic Level introduces objects and operations with which the user accomplishes his task. The *user interface* and its *implementation* part company at this level. Where the user speaks of *streams, units,* and *temperatures,* the implementation speaks of *data structures, subroutines,* and *variables.*

The user can do only those tasks which can be expressed in the set of objects and operations provided in the Semantic Level. The user is constrained to operate in this abstract universe. A common mistake of programmers is to violate the user's abstraction in one of two ways:

1. **Details of the implementation intrude.** Flagrant examples are invariably found among error messages of programs. The user is simulating a flash unit (an object from the Semantic Level), when the inscrutable message

   Secant method failed. ERROR=1.234E-5, ITER=16

   appears. The user has been dealing with streams, units, enthalpies, and has no way to "decode" this statement unless details of both the flash algorithm and the secant algorithm have been included in the grammar of the Semantic Level.

2. **A task promised at the Task Level cannot be accomplished within the Semantic Level.** An example is FLOWTRAN's (implicit) claim that arbitrary flowsheets can be simulated, when, in fact, a thorough knowledge of special unit operations, the FORTRAN language, FLOWTRAN's data structures, and numerical techniques is required of the user.

Whether or not the programmer intended it, these details *are* (or become) part of the Semantic Level and make the translation of tasks into actions more difficult for the user. It is one answer to the programmer's lament, "It's obvious to me; why isn't it obvious to you?" The Semantic Level described in the manual may indeed be transparently obvious, but in order to use the system, the user must understand levels of abstraction below or outside the Semantic Level.

What then is the appropriate level of abstraction, the set of terminal symbols of the Semantic Level, that should be provided for engineering design? We feel that *equations* and *variables* constitute the best choice. These are the *de facto* symbols of engineering simulation. The serious engineer, when faced with questionable output from a simulator, invariably asks to see the source code. Why? To extract the equations that predicted the strange results, and thence to judge whether the equations adequately model the process of interest. The engineer may lack the expertise to make that judgment, but can determine that only after examining the equations. The engineer is willing to accept for the moment the results of numerical algorithms if the equations can be verified later by hand.

Systems that attempt to package equations in larger chunks invariably force their interfaces into unnatural contortions in order to allow access to the constituent equations and variables inside those chunks. The complexity of the Semantic Level of ASCEND-II was greatly increased in order to access variables inside variable packs. FLOWTRAN's "control loops" are another example. The complaint of Professor Ed Cussler comes to mind; to paraphrase, "I never have understood what you people mean by 'unit'. What is a unit?" If the interface provides the means to access and manipulate equations, our colleague's question has a simple answer: A unit is a (small) system of equations that relates a (small) set of variables. Otherwise, the answer must include a plethora of confusing detail: streams, algorithms, inputs, outputs, design variables, unit parameters, control loops, *ad nauseum.*

The recent advent of equation-solving flowsheeting systems, such as ASCEND-II, described in section 3 of this paper, lends strong support to our advocacy of systems of equations as terminal symbols. It finally seems practical to separate cleanly systems of equations from the algorithms used to solve those equations. This has wonderful implications for the design of the Semantic Level of the user interface: the algorithm becomes a distinct semantic entity which the user can understand independently of the equations being solved. No longer must the distinction between "wide-boiling" and "narrow-boiling" systems be part of the semantic description of a flash unit.

Syntactic Level

The Syntactic Level introduces notations to designate objects and operations in the Semantic Level; it is the traditional understanding of a grammar. *Commands* to invoke operations that take as *arguments* the *names* of objects all appear here.

The design of interactive systems too often begins at or below the Syntactical Level. The design of higher levels, the system of abstract objects that permits easy translation of tasks into sequences of commands and gives form and meaning to the use of a program, is absent or haphazard. Such systems are difficult to describe concisely and hence are difficult to learn.

Interaction, Spatial Layout, and Device Levels

The lowest levels of the CLG deal with sequences of keystrokes, arrangement of output into "pleasing" tables, character codes and the like. These topics are beyond the scope of this paper. [Reisner79] uses a formal grammar of the Interaction Level to evaluate empirically the user interface of a simple interactive program.

1.2.2. Help

Every part of the interface should assist the user in the operation of the program. Some parts, however, are intended solely to *explain* the program to the user. The usual vehicle for explanation is a user's manual. But beyond the passive, static manual, interactive programs offer the possibility of active, dynamic aid to the user. A simple form of active aid is the "prompt" of a "question-answer" style of dialog. When a program types

```
    Number of stages:
```

a manual to explain input ("Columns 12-15 of the third line specify the number of stages in the absorber") becomes superfluous.

A single user almost never explores all of the remote closets in a program of broad functionality. The complexity of the Semantic Level of any useful engineering tool — parameters, equations, algorithms — would tax the memory of anyone. Indeed, even the author of a program is unlikely to remember such details. *All users are novices with regard to some aspects of the program.*

Several variations for assisting the user have been reported. Most are more or less elaborate schemes for the display of "canned text" in response to commands from the user. See, for instance, [Robertson79, McDonald80, Stallman80]. The help facilities of ASCEND-I described section 2.3 are in this class. Rothenberg [Rothenberg75] devised a more active approach. His intelligent tutor monitors the commands and mistakes of the user; when the user requests help, the tutor chooses

— having considered such criteria as the kinds of mistakes the user has made, the class of commands used, and the experience of the user — a piece of canned text that is most likely to resolve the problem. It goes yet further: if the user appears to be having difficulty, the tutor intervenes, questions the user about his task, and suggests remedial commands. A special "tutorial" mode is provided by the tutor, wherein the user can safely try out cpmmands without damaging his environment.

1.2.3. Participation

We have already noted that the engineer is expert at the Task Level, far beyond the abilities of any program. But implicit in the design of many CAD tools seems to be the attitude "Gimme a definition of the problem, and then get out of my way." The consequence of such arrogance is, in the case of flowsheeting systems at least, too frequently a failure to converge. Interactive computing offers a chance to tap the vast expertise of the engineer. The engineer may be able to detect divergence and terminate or redirect an algorithm. ASCEND-il contains the seeds for a Semantic **Level** rich enough to allow the engineer to participate in an overall "algorithm".

2. ASCEND-1: A Desk Calculator for Process Design

ASCEND-I is a "desk calculator" useful for the preliminary design of separation processes. Instead of the arithmetic functions found on most calculators, ASCEND-I has- such buttons as "enthalpy" and "flash" that evaluate thermodynamic relations and solve equations that model standard unit operations. These functions allow the user to investigate by case study the behavior of portions of flowsheets. The units cannot, however, be assembled into flowsheets. ASCEND-I does not perform the recycle computations performed by flowsheeting systems.

The program was developed for use by undergraduates in the process design course at Carnegie-Mellon University. Students in the course have completed prerequisite courses in thermodynamics and unit operations, but usually have little experience in computing. A single lecture introduces the program to the class. The lecture gives instructions for accessing the program and a general idea of what the program can do, but does not convey details of operating the program. Instead, the lecture explains how to use the "help" features of ASCEND-I. Students are expected to teach themselves by perusing on-line documentation and by experimenting with commands to learn their effects. We prefer to devote lectures to the application of the tool rather than the mechanics of its use.

2.1. The Semantic Level: The Abstract World of ASCEND-I

The user of ASCEND-I manipulates eight types of object. A brief description of each follows.

- A component is a collection of physical properties — *e.g..* molecular weight, critical properties, Antoine coefficients — associated with a particular chemical species. Binary properties; such as Wilson interaction coefficients, are associated with two components.

- Dictionaries are files that contain lists of synonyms for each component. **The** user identifies a component with any of several synonyms.

- Libraries are files of components.
- A system is a particular, usually small, set of components.
- A stream is a system, together with temperature, pressure, and composition.
- An experiment is a set of data that represents the measured behavior of a system under experimental conditions. ASCEND-I allows the user to fit parameters belonging to models to experiments. Only binary equilibrium data are treated in the current version.
- A lab is a file of experiments.
- A model specifies a set of equations that models a physical process, and an algorithm for their solution. Models are described in more detail below.

The operations on most objects are suggested (to users expert in the Task Level of process engineering) by their names and definitions: components can be saved, replaced, or retrieved from libraries; experiments and labs are similarly related. Synonyms, components, systems, and models can be defined.

2.1.1. Models

The ASCEND-I *model* is adapted from the *routing plex* of the ASPEN system [Evans77, Seider79]. A model is a hierarchical structure that reflects logically the "natural" organization of engineering correlations: any one of several correlations can be used to estimate a given property; also, a correlation for one property in turn requires the estimation of other properties. A logical *or* node in the model tree represents the options in the first case; an *and* node, the second. For instance, among the variables in the single-stage flash equations are equilibrium constants, liquid enthalpy, *and* vapor enthalpy. Vapor enthalpy may be written in terms of a perfect-gas enthalpy *and* a departure function. Enthalpy of departure can be estimated from any equation of state; in ASCEND-I the user may choose the perfect gas equation, the truncated virial equation, *or* a modified Redlich-Kwong equation. Figure 2-1 portrays a partial outline of a flash model.

A model is a convenient abstraction for several reasons:

- The hierarchical structure parallels both the user's organization of engineering equations as well as the organization of subroutines in the implementation.
- The structure is highly modular, and thus permits new correlations to be introduced with little revision of the existing code. This is a great advantage when the program is reconfigured for each year's design course.
- It includes under a single abstraction the entire functionality of the program. The user knows that all results are obtained by defining and running models. If he knows how to run one model, he can infer how to run another.

FLASH

• ***K-VALUE***

. . . **(submodels of *K-Value*)**

* ***LIQUID*** *ENTHALPY*

. . . (submodels of *Liquid Enthalpy*)

* ***VAPOR*** *ENTHALPY*

* *PERFECT-GAS ENTHALPY*

* ***VAPOR*** *ENTHALPY OF DEPARTURE*

+ *PERFECT* *GAS ENTHALPY* ***OF*** *DEPARTURE*
+-J&UNCATED *V/R/AL ENTHALPY OF DEPARTURE*
. . . Model for second virial coefficient

+ *REDUCH-KWONG ENTHALPY OF DEPARTURE*

Legend

• denotes *and:* all of the submodels must be specified
+ denotes *or:* exactly one of the submodels must be specified

Figure 2*1: Partial outline of a flash model

- **It serves a pedagogical function: the connections between the many thermodynamic quantities are plainly revealed.**

Associated with each type of model are input parameters, output parameters, and algorithmic parameters such as convergence tolerances and iteration limits. A simulation is effected by running a model; at that time, the user specifies .the algorithmic parameters and input parameters with commands in a special "run mode" for that model. Many models have as parameters objects such as components, systems, and streams. Sections in the on-line documentation {see section 2.3) list the equations used in each type of model, their assumptions and limitations, and explain details of iterative algorithms. Command modes for declaring models and for running models contain pointers to these sections of the documentation.

Models can be "traced" so that, as iterative calculations proceed, intermediate values can be observed. For instance, activity coefficients can be observed to see if they display erratic behavior during a flash calculation.

Models can also be run separately. Fugacity coefficients, molar volumes, flashes

— all *csarr* be defined: ass models and: rurr irr the^ same manner.

Despite the attempts to make them as transparent as possible, the ASCEND-I models are very complicated abstractions. Not only must the user understand the many equations and variables involved (which is properly the grammar of the Task Level; that is; we expect the users to be competent process engineers); the user must as well understand: a? multitude of algorithms. A different algorithm must be provided for each flavor of flash calculation — isothermal, adiaixalic; vapor fraction — everr though the equations anct variables involved: are identical.

2.2. The Syntactic Level: Command Language

The user types coammrands to ASCENDH on a keyboard terminal. The program indicates when it is ready to process a command by typing a prompt at the beginning of at line; Commaaitts aree grouped: by *modes* according to function. For example, the several commands that definee ancfc modify systems ara ayailafcdee in "systwrr" made. The current* rmxdBE is indicates! by the prompt: A- spesiaJ command "cecit", moves? the use? frrmr thee current' motdeb inta theB previous" modfe

Most CEQTTmrarcte resemble impoerariveE or* d&tilarativeE sa8ntence5& For* example*

>HUK (moai^l) EL£S»

executes the model named: "FLASH".

2.3. Help

New users of ASCENEH a^e ©charted, encouraged, required: to teach thejnsedvess how to run the program. They are armed with two facts:

1. Every carTmraaid mode has a "help" command that can provide help on topics relevant far that mode.

2. Two rrragic keys caaose ASCBSIDH to e9cptaàn itself: orree for each of- two levels of *d&m*l. TheE firs^ level deals with the syntax of commands; the second accesses saructureact UuCinnBiualiarr sdarefi: arr tesct*. filesst . -

2.3.1. Syntactic Help

The first level of help is a feature of the TOPS-20 command parser COMND [DEC80]. At any point in a command line, if the user types a question mark, COMND responds with a list of legal symbols for that portion orf the command. Thus the user does not have to remember, for instance, what commands are^ a^arilairiee in a command mode, *or* the options of a particular command.

2.3*2. Semantic Help: The OrHUne User's Manual

Complete documentation of the program is kept on disk files. A separate program, called DOC, presents the files to the user in the familiar guise of a book, sectioned in the form of an outline.. The tap level is section number zero; under it am OTStierrs T*. Z i. ^.;; uwttrirr gpgctian Z are SBesctiansE ZT, ZZ Z3t. ^; aficb sEOT forth; Thee iwwr ©EHT move between sections in saequemial uitlBf (Z Z1; ZT.r. Zi:Z Z2J with TMexf »id "Previous" commands (by striking: keys *N* or *P);* "Ujr" to theE superior section (Z.T.Z, ZT) *or* "Dawn" ta anr inferior (ZT, ZT.2);; "Farwartf' and: "Back" atf thee same level (2, 3), or can "Go" to any section by typing its number. "Footnotes" cross-reference sections of the documentation. When the user "arrives" at a section. its text is displayed on the terminal. With the "Contents" command, he can see the titles of sections inferior to the current section.

If, instead of a question mark, the user strikes a special "help" key on the terminal, this DOCument reader is called. In the source code, the program supplies the command parser with pointers to the documentation, so when the "help" key is pressed the user jumps immediately to that section of the documentation that describes the particular field of the command being typed. Once "inside" DOC, the user can move to the adjacent sections (that presumably cover closely related subjects) if the initial section didn't answer the question.

The rationale for such a documentation system is that maintenance of a program must be directly linked to its documentation: as the interface is modified, so must the documentation be revised. The help pointer in the source code makes this link explicit. Distribution of the most current documentation is unnecessary, since it is always available for immediate access via the command language.

3. ASCEND-II: an Interactive Flowsheeting Package

3.1. Overview

Whereas ASCEND-I can only solve individual models, corresponding to thermodynamic calculations and individual unit operations, the ASCEND-II program [Locke8i] is a flowsheeting system [Westerberg79]. The user can construct a model for a chemical process comprising a number of process units arbitrarily interconnected. It is an equation-based system; thus the building blocks are equation sets which are collected together in response to the user input defining the process to be modeled. Once all the equations are gathered together (there could be thousands), the user can specify which variables are to be fixed and which are to be calculated, and then ASCEND-II automatically develops a solution procedure for the entire set using the Newton-Raphson technique and sparse matrix methods augmented with decomposition strategies [Westerberg78], [Clark80]. Thus ASCEND-II is remarkably flexible in the types of problems it can solve. All the variations of flash units (isothermal, adiabatic, etc.) are represented by a single equation set but with different variable sets selected as fixed and calculated.

This flexibility of ASCEND-II is both the source of its power and the potential source of its defeat. A serious consideration in developing ASCEND-II was to give the designer a means to exploit this flexibility without getting lost. To us, this requirement suggested a high level of user interaction and the ability to allow evolutionary model development.

In designing ASCEND-II, we envisioned a flowsheeting package that would allow the user to evolve towards the solution of his problem along three independent axes, as shown in Figure 3-1: a model complexity axis, a calculation type axis, and a computation detail axis.

In the direction of model complexity, the user can begin with simple models of

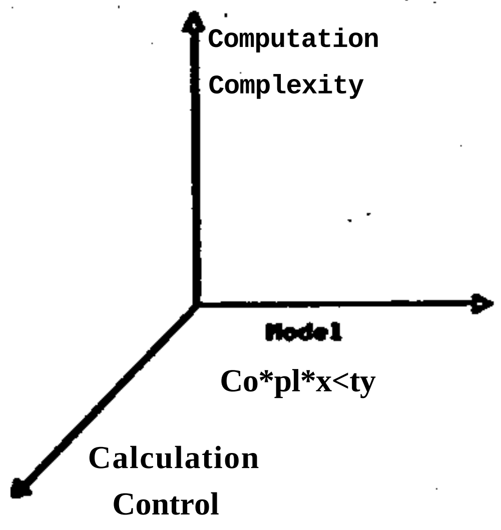


Figure 3-1: Modeling, Calculation, and Computation Axes

one or two process units. This model will (hopefully) converge quickly and can be **used** as **a** basis for further calculations. The user can then probe this model, for **example** by adding more complex physical property calculations. The converged **values of** the simplified model are used as a starting point for this more rigorous calculation. If the more complex physical property calculation is necessary, it can be **left** in. If not, the user can readily return to the simple calculation.

Another way of increasing model complexity is by adding more process units to a **flowsheet.** The user can add process units one by one, each time probing the **complexity** necessary in the physica! properties, until his entire flowsheet is modeled. **The** converged values obtained from an earjier flowsheet would in each instance be **used to** initialize the current flowsheet model. Proceeding in this manner, by making **small** moves, probing, initializing with a previous solution, helps to insure convergence at each stage. This capability also gives the user confidence in the **numbers** which the machine generates since he is able to re-examine the progress of **the** model at each stage.

The second axis in Figure 3-1 represents user control of calculation type. The **simplest** type of calculation performed by ASCEND-II is a simulation calculation. **This** is the type of calculation most* easily performed by sequential modular simulators. It[1] is also called a rating calculation. All flowsheet inputs and unit **parameters** are specified by the user. The system calculates unit outputs and **intermediate** stream flows. Because the user can more easily visualize that a unit **must** operate if its input streams, sizes, and operating levels are given, a simulation **calculation** is the type of calculation for which ones intuition is strongest. A person **will more** likely handle the degrees of freedom correctly in this mode.

A more difficult calculation is a design calculation. In this calculation, the user specifies an output variable value and asks the system to calculate a unit parameter or flowsheet input. It is more difficult in two ways:

1. The user may not specify easily a proper set of variables, leaving a singular set of equations.

2. It is possible to put a specification on the outlet stream which is impossible to meet, in which case the equations do not converge or they converge to a nonsensical solution, and the user is left wondering what happened.

A simple example of this second case is a distillation column. In the simulation calculation, the user specifies the feed, feed tray location, reflux rate, and number of **trays.** The simulator calculates distillate and bottoms stream compositions and internal stream compositions. Given the column inputs and unit parameters, the user is confident that outputs can be calculated. In a design calculation for a column, the **user** may specify the feed, feed tray location, number of trays, and a mole fraction in the distillate. The program would calculate reflux rate and all unspecified stream **flows** and compositions. The problem here is that the number of trays specified may be below the minimum number needed to perform the required split. No value for reflux rate can be calculated under these conditions, and the equations will not converge.

The correct way to solve the above problem using ASCEND-II is to start out with a simulation calculation; i.e., specify reflux rate, number of trays, etc. to get a converged solution to use as a starting point. Then, one can adjust the reflux rate and number of trays until the target distillate mole fraction is close to its desired value. Now one can switch to a design calculation, specifying the mole fraction and releasing the specification on the reflux rate. The equations should converge quickly to a meaningful solution.

The current version of ASCEND-II is able to perform simulation and design calculations. Kuru [Kuru8i] has laid the groundwork for adding dynamic simulation. In the future, optimization capabilities, based on a modification of the Han-Powell algorithm, will be added [BernaSO]. These capabilities will extend the calculation axis to include dynamic simulation and optimization.

The idea of progressing from a simple to a more difficult calculation still holds. A user should not rush in and try to perform an optimization or dynamic simulation. He-should first become familiar with his flowsheet by performing a simulation. * If performing an optimization, he should move the specified variables around to see the effect on the objective function. This way he will have some idea of what should occur when he finally runs the optimization.

The third axis in Figure 3-1 represents user control of the computation sequence. At the lowest level the user would be totally naive of the details. We have provided a disc file containing the commands necessary to solve a problem. The naive user **would** tell ASCEND-II to read commands from this file.

An experienced user would exercise more control over the details of the computation. Rather than telling ASCEND-II to read commands from a file, he would give the commands himself interactively. He can input his flowsheet interactively and give the commands to convert it to executable form. Once this is done, he can interactively input needed initial values and give the command to initialize the rest of the flowsheet variables based on this input, some of which may be guessed. Now he is ready to solve the equations. He may wish to perform an iteration of the solution algorithm and then rescale the variables and equations. If he is having trouble converging, he may isolate the troublesome unit model and work only it for a few iterations. He gives these commands interactively, in effect controlling the details of the computation. An analogy might be the driving of a car where the user learns about his problem and how to maneuver around the obstacles. It can be a task done well even without ones knowing details on how the engine and transmission work.

3.2. Two-stage Input

ASCEND-II is an equation solver. The executive system does not know which kinds of equations it is solving. ASCEND-II could just as well solve civil or mechanical engineering problems.

The first phase of the input is contained in a file which the casual user never sees. This information sets up the environment for chemical engineering models. It goes hand in hand with the subroutines which model the units. The details on this input are explained in [Locke81].

The second phase of input includes the user's flowsheet, initial values, and specifications for fixed and calculated variables. This information can be input from a file or interactively. User input is simplified through the use of Macro Expansion and Default Cards [Locke80]. With Macro Expansion the user can create complicated models through simple input. Figure 3-2 shows how Macro Expansion works for a distillation column. The user specifies that he wants a distillation column. He names the inlet and outlet streams of the column, number of stage models top and bottom, and a physical property option. The distillation column "expands" into two column sections and the mixers, splitters, flash, and reboiler necessary to do the modeling. Each column section further expands into the individual stages which model the column. Each stage then expands to include the desired physical property calculations for enthalpy, entropy, and K-values.

Default Cards are a convenient way to establish the same information for several unit models. Instead of putting the same words on several cards, the user puts the name of a default containing those names. Default Cards are conveniently used with Macro Expansion to specify physical property options.

3.3. Solving Equations Using ASCEND-II

ASCEND-II uses Newton-Raphson to solve equations. All equations are written in the form f(x,u)=0, where the u variables are the user selected fixed variables and the x variables are those remaining to be calculated. A first order Taylor series expansion of this equation is:

$$\underline{f}(\underline{x}+\Delta\underline{x},\ \underline{u}+\Delta\underline{u}) = \underline{f}(\underline{x},\underline{u}) + (\delta\underline{f}/\delta\underline{x}^T)\Delta\underline{x} + (\delta\underline{f}/\delta\underline{u}^T)\Delta\underline{u}$$

where $(\delta\underline{f}/\delta\underline{x}^T)$ and $(\delta\underline{f}/\delta\underline{u}^T)$ are Jacobian matrices (matrices of first order partial derivatives). Since the $\underline{u}$ variables are fixed, the ASCEND-II system eliminates them

MfiCRQ EXPANSION

Complex Models From
Simple Input

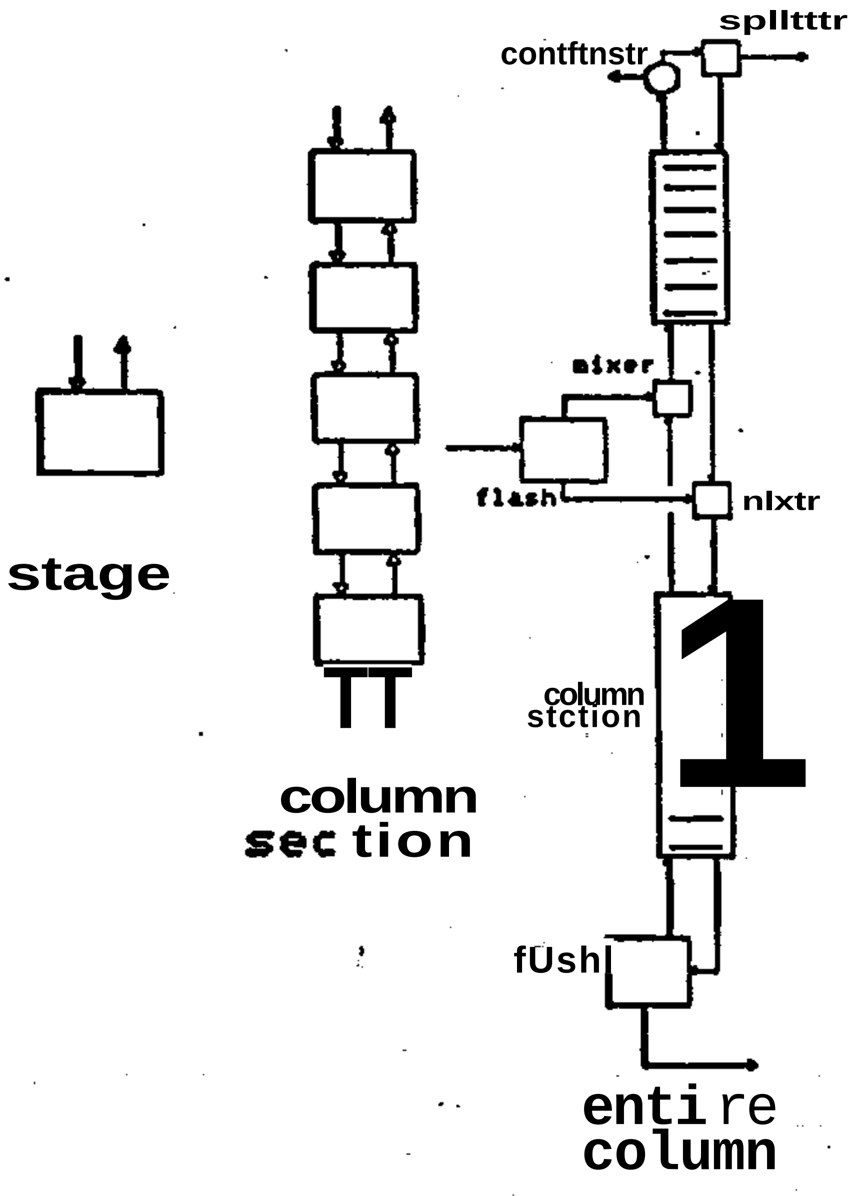


Figure 3-2: Macro Expansion for a Distillation Column

from the problem, and the equation becomes:

$$\underline{f}(\underline{x}+A\underline{x}) = \underline{f}(\underline{x}) \text{ «• } U\underline{f}/3\underline{x}^T)A\underline{x}.$$

Setting $\underline{f}(\underline{x}+A\underline{x}) = \underline{0}$ leaves the linear system of equations:

$$(i\underline{f}/5\underline{x}^T)A\underline{x} = -\underline{f}(\underline{x}).$$

This system of equations is solved repeatedly until the norm of vector $\underline{f}(\underline{x})$ (the negative of the residual or Right Hand Side (RHS) vector) is less than some small **tolerance.**

For the norm of the RHS vector to be meaningful all equations must be scaled to the same order of magnitude. Material balance equations may contain terms on the **order of** 100. If a material balance equation were off by 1%, its residual would be 1.0. An equation summing mole fractions has terms on the order of 1.0. If off by 1% this equation has a residual of .01. In order to compare these equations, the material balance equation would be scaled by dividing by a factor of 100. Variables **are also** scaled in order to keep the Jacobian matrix well conditioned.

The subroutines which calculate scaling factors can be called at any time during the solution procedure. If an initial guess is far from the actual solution, variable values could change by several orders of magnitude. We have often found that a group of converged equations may appear not to have converged if the scaling factors are far **off.**

One could look at ASCEND-II as a flowsheeting system which allows the user to **perform** several operations on a file of steady state data. This is shown in Figure 3-3. Besides scaling equations, the user can initialize variables, attempt to solve the equations by performing one or more Newton-Raphson iterations, and change degrees of freedom, all at any time, interactively.

Initialization subroutines exist for each type of unit. These subroutines calculate outlet stream and internal variable values based on guesses for inlet stream variables. They work in much the same way as sequential modular models except no equations are converged. The user can initialize the entire flowsheet at once or single units, one at a time.

When solving a flowsheet, the user has similar options. He can specify that the entire flowsheet be solved simultaneously or he can solve individual units or groups **of** units.

Because we use Newton-Raphson to solve equations, changing degrees of freedom **is easy** with ASCEND-II. This change amounts to exchanging a fixed (u) variable for **a calculated** (x) variable. Different columns are included in the Jacobian Matrix when **degrees** of freedom are changed.

3.3.1. Report Generation

Figure 3-4 shows schematically the report generation facilities of ASCEND-II. **ASCEND-II** includes a detailed pointer system which allows the user access to **information** about his flowsheet description, variable values, fixed/calculate flags, and **equation** residual values. On-line symbol tables allow equations and variables to be

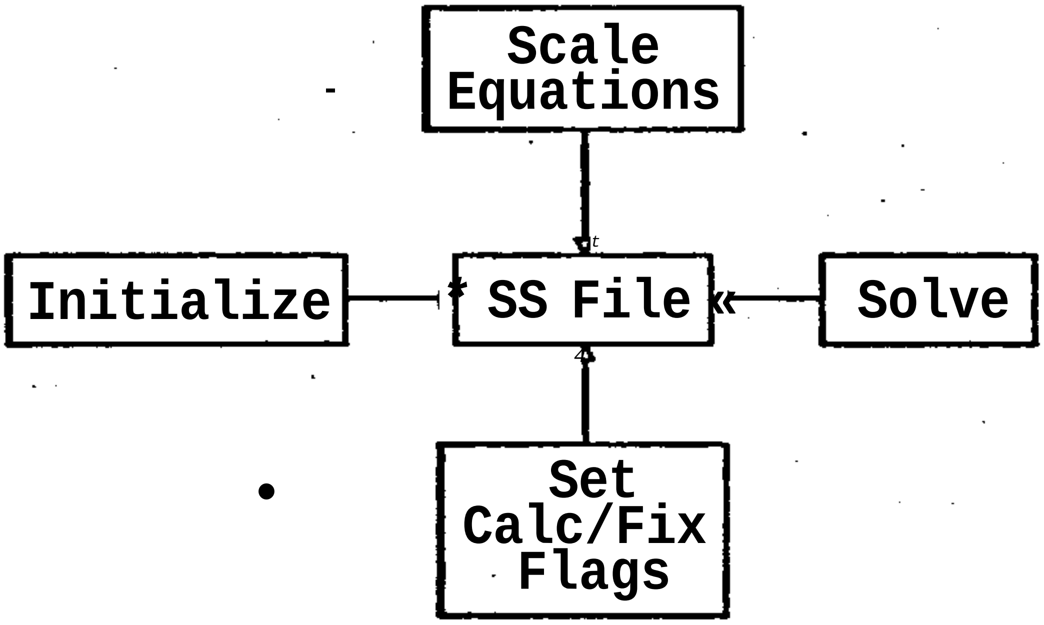


Figure 3-3: Operations on a Steady State File

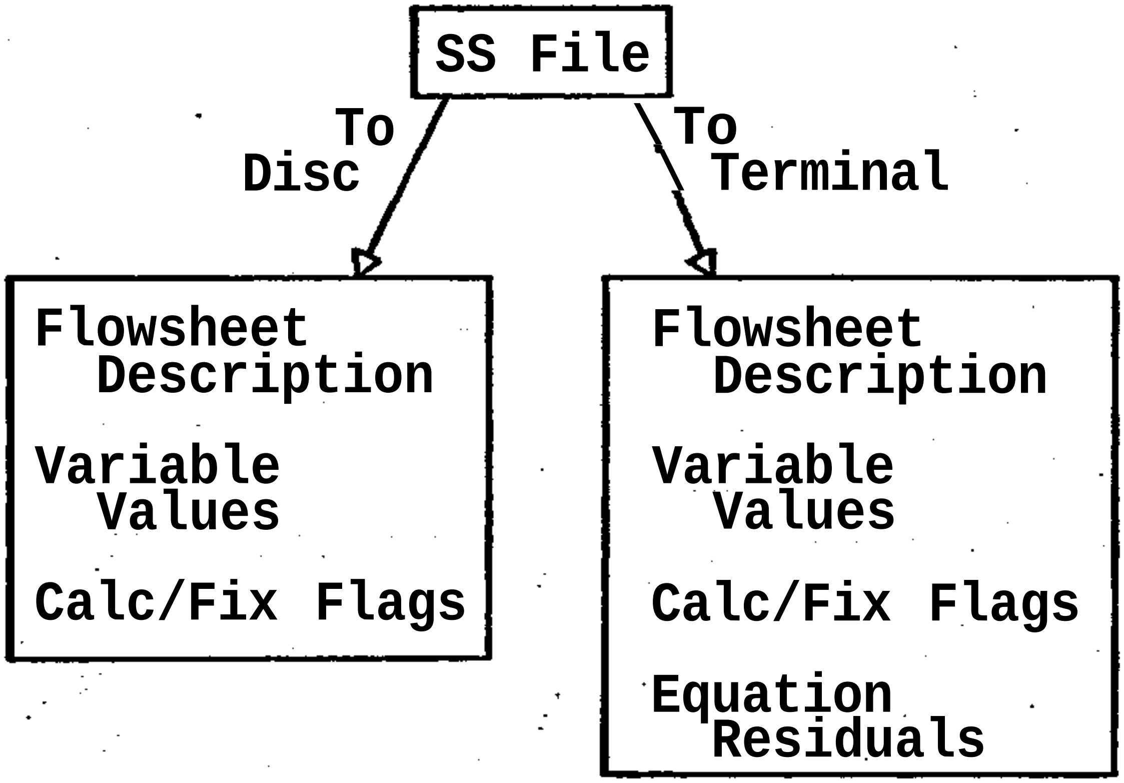


Figure 3-4: Report Generation Schematic

accessed by name at any time.

The flowsheet description (original or expanded) can be displayed on the terminal to be verified by the user or can be saved on disc to be read in later. Variable values and fixed/calculate flags can also be displayed or saved on disc. Scaled equation residuals can be displayed at the terminal, unit by unit. They show how far a unit's equations are from converging.

3.3.2. Solving Flowsheet Equations

After the user inputs his flowsheet description, he initializes (guesses) input and recycle streams, then gives the command to initialize the rest of the flowsheet. Once he has issued the command for the system to scale • the equations and variables, he is ready to solve the set of equations. If he has a good initial guess, the equations will converge quickly . The engineer can make use of the tools we **have** provided for him if he has trouble converging. The first thing to do is to print out the equation residuals. This step tells the user which units have not converged. He could then isolate this unit from the rest of the flowsheet. Perhaps he would re-initialize this unit, then rescale the variables and equations. He could then solve this unit's equations by themselves, separate from the rest of the flowsheet. Once the troublesome unit had converged, it could be merged with the rest of the flowsheet and he could attempt again to solve the whole system simultaneously.

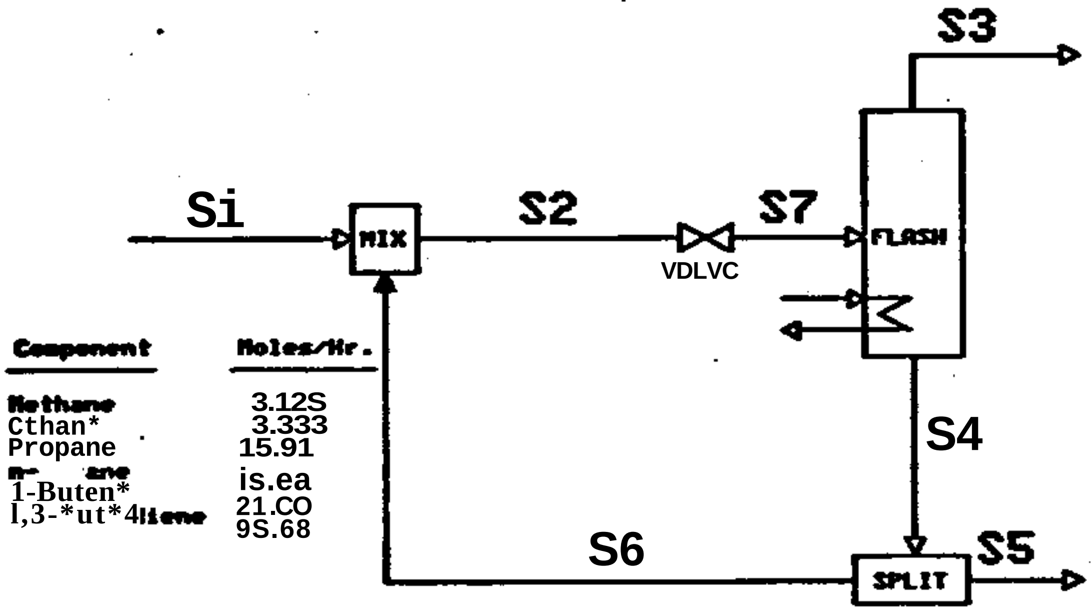
3.3.3. Solving a Large Flowsheet

Newton-Raphson will quickly converge from a good initial guess. The problem is to get a good initial guess. Initialization subroutines help to do this. For a large flowsheet, this may not be enough. The user can then evolve his flowsheet from a simple start to the complex detail he really wants. Three ways to do this will be illustrated with the flowsheet of Figure 3-5, a flash unit with recycle.

The first method is to run the flowsheet with only three or four of the most plentiful components. In this flowsheet, the user would first include the heavy components, propane, n-butane, 1-butene, and 1,3-butadiene. This cuts down substantially on the number of equations which must be solved. The K-values in the flash and the temperatures in the flowsheet should converge close to the values for the solution of the full problem. The user would save these values and modify the flowsheet by adding-~the rest of the components. He would then initialize the full flowsheet based on the values of the scaled-down version. These initial guesses should be close enough to converge the equations.

Another way of evolving a flowsheet is to add the pieces of equipment one at a **time.** The use would start with the flash and converge it, using as its feed stream S1 plus an estimate of stream S6. Then he would add the splitter and converge the flowsheet, again using saved values to initialize the flash unit. Finally he would add **the** mixer and valve, once again initializing with the previous flowsheet's values, and converge the entire flowsheet. Since the user can solve parts of a problem individually, he could input the whole flowsheet once and solve it piece by piece without having to re-input the flowsheet at each step.

The most successful way to evolve a flowsheet is to start with the assumption of **ideal** thermodynamics and not to do an energy balance. Temperatures would not be **calculated** in the initial pass, nor would nonideaiities contribute to the K-values in the



Component	Moles/Hr.
Methane	3.12S
Cthan*	3.333
Propane	15.91
[illegible]	is.ea
1-Buten*	21.CO
l,3-*ut*4iene	9S.68

Figure 3-5: Flash Unit with Recycle

flash unit. The overall material balance, however, would likely be very close. Removing the rigorous physical property calculations cuts the number of equations by nearly two-thirds. This also eliminates the highly nonlinear thermodynamics equations which are most difficult to solve. After converging the simplified flowsheet, the user would turn on the rigorous calculations, and initialize the flowsheet with the simplified model. This should get him close enough to converge **the** equations quickly.

3.4. Future Work

The next step in the development of ASCEND-tl is to add the optimization capability. This work is presently under way and should be complete within a year. As was mentioned previously, a modification of the Han-Powell algorithm will be used. This fits in well with the Newton-Raphson scheme, as the linearized equality **and** inequality constraints necessary for the optimization will be produced by the **same** subroutines which generate the partial derivatives and residuals for solving the **equations** using Newton-Raphson.

The optimizer would allow the user to choose any arbitrary variable as the objective function. The user could, for instance, choose to minimize the mole fraction of an impurity in a product stream subject to constraints on equipment **costs.** Another use would be to fit data such as Wilson parameters. In this case **the** objective would be to minimize the sum of the squares of the errors in the **fitting** equations.

Dynamic simulation capability will also be added to ASCEND-II. The change from steady state to dynamic simulation turns out to be relatively minor if the equations are posed correctly. Discretization of the differential equations converts them to algebraic equations which can be solved using ASCEND-!I. ASCEND-li dynamic simulation is discussed in detail in [Kuru8i], and also in [LockeSO].

The user oriented ideas of ASCEND-I will also be implemented in ASCEND-II. The friendlier input language and the on-line documentation as used in ASCEND-I will be added to ASCEND-II.

References

[Berna80] Berna, T.J., M.H. Locke, and A.W. Westerberg.
A New Approach to Optimization of Chemical Processes.
AIChE J. 26(1):37, 1980.

[Clark80] Clark, P.A.
An Implementation of a Decomposition Scheme Suitable for Process Design Calculations.
Master's thesis, Carnegie-Mellon University, August, 1980.

[DEC80] *TOPS-20 Monitor Calls Reference Manual, AA-4166D-TM*
Digital Equipment Corporation, Marlboro, Massachusetts, 1980.

[Evans77] Evans, L.B., B. Joseph, and W.D. Seider.
System Structures for Process Simulation.
AIChE Journal 23(5):658-666, September, 1977.

[Kuru8ij Kuru, S.
Dynamic Simulation With an Equation Based Flowsheeting System.
PhD thesis, Carnegie-Mellon University, 1981.

[Locke80] Locke, M.H., S. Kuru, P.A. Clark, and A.W. Westerberg.
ASCEND-II: An Advanced System for Chemical Engineering Design.
In W.G. Vogt and M.H. Mickle (editors). *Modeling and Simulation Volume 11.* pages 409-413. Instrument Society of America, 1980.
Presented at Eleventh Annual Pittsburgh Conference on Modeling and Simulation.

[Locke81] Locke, M.H.
CAD Tools Which Accommodate an Evolutionary Strategy in Engineering Design.
PhD thesis, Carnegie-Mellon University, 1981.

[McDonaldSO] McDonald, P.H., and T.J. Thompson. .
Designer's Workbench: The Programmer Environment.
The Bell System Technical Journal 59(9):1793-1809, November, 1980.

[Moran78] Moran, Thomas P.
Introduction to the Command Language Grammar.
Technical Report SSL-78-3, XEROX Palo Alto Research Center, October, 1978.

[Reisner79] Reisner, Phyllis.
Using a Formal Grammar in Human Factors Design of an Interactive Graphics System.
Technical Report RJ2505(32755), IBM Research Laboratory, April, 1979.

[Robertson79] Robertson, G., D. McCracken, and A. Newell.
The ZOG Approach to Man-Machine Interaction.
Technical Report CMU-CS-79-148, Department of Computer Science, Carnegie-Mellon University, October, 1979.

[Rothenberg75] Rothenberg, Jeff.
An Intelligent Tutor: On-line Documentation and Help for a Military MessageService.
Technical Report ISI/RR-74-26. University of Southern California, Information Sciences Institute. May, 1975.
Distributed by National Technical Information Service.

[Seader74] Seader, J.D., W.D. Seider, and A.C. Pauls.
FLOWTRAN Simulation: An Introduction -CACHE Committee.
Ulrich's Books, Ann Arbor, Michigan, 1974.

[Seider79] Seider, W.D., L.B. Evans. B. Joseph, E. Wong, and S. Jirapongphan.
Routing of Calculations in Process Simulation.
Ind. Eng. Chem. Process Des. Dev. i8(2):292-297, 1979.

[StallmanSO] Stallman, Richard M.
EMACS, the Extensible, Customizao/e, Self-Documenting Display Editor.
Al memo 519, Massachusetts Institute of Technology, 1980.

[Westerberg78] Westerberg, A.W. and TJ. Berna.
Decomposition of Very Large-Scale Newton-Raphson Based Flowsheeting Problems.
Computers and Chemical Engineering 2(1):61-63, 1978.

[Westerberg79] Westerberg. A.W., H.P. Hutchison, R.L. Motard, and P. Winter.
ProcessFlowsheeting.
Cambridge University Press, 1979.