A More Reusable Modeling System

Benjamin Andrew Allan

April 3, 1997

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the department of Chemical Engineering at Carnegie Mellon University.

Arthur Westerberg, Advisor

Examination Committee:

Lorenz Biegler, Dept. of Chemical Engineering

Susan Finger, Dept. of Civil Engineering

Spyros Pandis, Dept. of Chemical Engineering

Arthur Westerberg, Dept. of Chemical Engineering

ABSTRACT

In this thesis we examine problems in the reuse of mathematical models. Reuse of a model by others requires that the model is viewable from many different perspectives, that the model is capable of incorporating new information during its evolution, that the model is understood by a range of modelers in different times and places with different special skills, that the model is mechanically computable in many different ways, that the model's assumptions and key data are not lost, and that the model can support conflict resolution when it is reused as part of a larger model.

Our hypothesis is that the problems from which these requirements arise can be overcome, leading to a significant improvement in future model building practice. We here assemble in an original way a group of concepts from engineering design, mathematics, and computer science which, taken together, make model reuse much easier to achieve. We present a modeling environment, ASCEND IV, which we have used to test the our concept.

We take a hard look at widespread concepts of object-oriented and modular software reuse and find that they cannot efficiently meet the requirements of reusable modeling. We suggest that, even in an ideal world, there is no single tool which can meet all these requirements and support all users well. We suggest that computer tool makers should design future computer tools around information sharing rather than information hiding, so that tool users can combine different tools and models to attain their ends efficiently.

We have constructed an example of an open tool for large-scale modeling work, ASCEND IV, which incorporates an equation-based modeling language derived from ASCEND IIIc and concepts similar to the "design by contract" concept presented in the Eiffel language. Our system also features an open user interface and an extensible solver architecture for supporting interactive modeling with nonlinear, logical, and structural constraints and for supporting peer-to-peer interaction with other tools.

Acknowledgments

I am deeply indebted to Thomas Epperly, Peter Piela, Arthur and Karl Westerberg for creating the ASCEND modeling language concept, and to Kirk Abbott and Joseph Zaher for going with me far beyond the bounds of conventional research to create the ASCEND IIIc modeling environment. Robert Huss, Jennifer Perry, Vicente Rico-Ramirez, Boyd Safrit, Mark Thomas, and Kenneth Tyner have all been very influential in the design and implementation of the modeling language, models, and tools presented in this thesis.

There is a cast of hundreds spread among the Department of Chemical Engineering, the Engineering Design Research Center, the Norwegian Technical University at Trondheim and the Computer Aided Process Design consortium member companies to whom I am grateful for the roles they have played as patient system testers, anguished modelers, design debaters, provocateurs, and friends. These have helped me discover a systems research that is much more fruitful because it acknowledges the primacy of humans in the systems.

This work is dedicated to the future users of mathematical modeling systems and to my parents, Andrew and Emily Allan, and family.

Table of Contents

1	Modeling is hard because reuse is very hard	1
1.1	The problem	1
1.2	Our solution	4
1.3	Open modeling issues	6
1.3.1	Sharing common information	7
1.3.2	Anonymous types	8
1.3.3	Open objects and “modeling by contract”	10
1.3.4	Intellectual property and openness	11
1.3.5	Attacking the puzzle	12
2	Changing Modeling Systems	14
2.1	Design	14
2.2	Open form modeling problems	16
2.3	The modeling systems landscape	18
2.4	Open form modeling solutions	21
3	An expanded view of component-based modeling	28
3.1	A perspective on current software	29
3.2	Components in process modeling	33
3.3	An open form model component definition	37
3.3.1	Examples of desired component reuse	38
3.3.2	Design properties	40
3.3.3	Model component attributes and messages	41
4	Toward better modeling languages	52
4.1	An open form modeling system	52
4.2	Problems in mathematical modeling tools	54
4.3	Problems in ASCEND III	57
4.3.1	The properties of ASCEND III	58
4.3.2	Why reusing ASCEND III is hard	63
4.4	Desirable language properties	65
5	ASCEND IV Modeling Language	70
5.1	User speed and syntax consistency	71
5.2	Clearly defined operators	73
5.3	Efficiency	80
5.4	Extensibility	87
5.5	Modeling by contract	90
5.6	Language results	99
6	Tools to support modeling with objects and equations	103
6.1	Helping the user manipulate model information	103
6.1.1	Parser and compiler messages	103
6.1.2	Refinement hierarchy	104
6.1.3	Information hiding	105
6.1.4	Where is this created?	106
6.1.5	Reusable dynamic modeling	107

6.1.6 Model reordering 107
6.1.7 GAMS code generation 107
6.2 Immediate extensions based on this work 108
6.2.1 Constructed explanation of names 108
6.2.2 Cut and paste modeling 108
6.2.3 Classification of objects 108
6.2.4 Beefing up ASCEND IV methods 109
6.3 Future extensions 110
7 Summary and conclusions 112

List of Code Examples, Figures and Tables

Figure 1-1: Merging stream Unit2.out with Unit1.in for Flowsheet1 7
Figure 1-2: Passing a common stream to both units through interfaces 8
Figure 3-1: One engineering modeling system architecture 33
Figure 3-2: A General Modeling Environment 34
Figure 3-3: Production planning model (A) constructed from process design (D) and control (B) models. Reactors in (B) are defined including parts of the chemist’s bench reaction-separation model (C). 40
Figure 3-4: Reusable model and solution tools information flow. 43
Table 3-1: Desirable information in a reusable mathematical model 44
Figure 4-1: Part-whole structure of a distillation column. 63
Figure 4-2: A connection between self-contained objects 64
Code 5-1: MODEL pipe_flow; 74
Code 5-2: MODEL rxntest; 76
Code 5-3: `FOR i IN [1..10] CREATE` 77
Code 5-4: MODEL liquid_mixture(78
Code 5-5: MODEL td_equilibrium_mixture (80
Code 5-6: MODEL alias_example; 81
Table 5-1: Some names of water property data 82
Table 5-2: Merge statistics for a 13 tray methanol/water column 82
Code 5-7: MODEL euler_integration(83
Table 5-3: Isomerization2 collocation model compilation times 84
Code 5-8: MODEL liquid_feed_tray(88
Code 5-9: ADD NOTES FOR TYPE liquid_feed_tray; 89
Code 5-10: MODEL pump; 90
Code 5-11: MODEL hexane_column_flowsheet; 93
Code 5-12: MODEL test_vapor_liquid_flash; 97
Figure 6-1: Library refinement ancestry (class hierarchy) display tool 105
Code 6-1: METHOD init_trays(status WILL_BE symbol); 110
Table 7-1: Reusable modeling problems and solutions 112

CHAPTER 1 MODELING IS HARD BECAUSE REUSE IS VERY HARD

1.1 THE PROBLEM

The design, construction, operation, retrofit, and decommissioning of a chemical process plant are done by diverse teams of specialists over long periods of time. These specialists are increasingly likely to be spread over large geographic distances and across corporate boundaries. These specialists use a wide variety of currently incompatible tools to create the many different kinds of information (models) needed to support decision making and record keeping. For example, a chemist is likely to use a statistics package or a sophisticated chemistry modeling software, a design engineer is quite likely to use a spreadsheet to define and solve preliminary material and energy balances based on rather limited information, a control engineer may use a dynamic flowsheet simulation tool, and a process operations planner is likely to use a rather different spreadsheet model or an equation-based optimization language.

At almost every stage, specialists are recreating models because it is inconvenient or impossible to reuse the information that went into constructing the models of other specialists on the team or to reuse the models of previous teams. Even if the various specialists rigorously abide by standards[1] and their software tools are remarkably more compatible than are current tools, there are still many hurdles which can block successful sharing of models and modeling tools. Reinventing the wheel is still a problem.

This problem is not unique to process plant modeling; it also affects research in engineering science, methods, and systems. An enormous literature exists which documents much progress in the invention of new or improved solution techniques for a wide variety of mathematical problems. Application of these new algorithms to particular physical systems has created new fields of investigation, for example, molecular dynamics and molecular design, and has increased the range and detail of existing models, for example, the constantly improving methods for estimating the thermodynamic properties of mixtures. Challenging applications drive the invention of new algorithms in a feedback loop. This research progress seems all the more remarkable when we consider that many research students do not effectively reuse tools (experimental apparatus or software) or knowledge created by previous students working on the same or similar projects.

Relatively little work has been published on the very hard problem of building a unifying conceptual framework that will enable a diverse team of technical specialists performing pieces of process modeling to absorb efficiently the wealth of ideas and tools that come out of software companies and universities or to combine their efforts easily with those of other teams. This missing framework also affects the university research project itself, inasmuch as researchers find it difficult to share (or to bear the cost of) anything other than the simplest of computational tools and models.

We can identify many problems in reusing the work of others, even when they use the same set of computational tools that we do.

1. Varying views of model structures and data are required for different specialists and different tasks. What these views will be is not always easily anticipated, in part because the boundary between specialties is less and less clear.
2. Initial modeling must be done with very limited information. Many tools make it diffi-

1. This is unlikely, since each specialty has it own tools and jargon and since process modeling work frequently requires unique models to be produced on very tight schedules.

cult to incorporate subsequent information and evolve the initial model.

3. Communicating the expected and potential uses of a given model across barriers of time, place, and specialty is hard. Most tools cannot capture information about expectations and limits of application.
4. Finding the commonalities among different modeling paradigms[2] to allow computational support for model sharing is difficult.
5. Tracking model data and model assumptions through time to allow recovery from modeling errors or to help define new modeling directions is not supported by most tools.
6. Negotiating conflicts when combining the efforts of different modelers must be performed. Even the best organized people may need incompatible assumptions while working on separate subproblems.

Considering the above list, it is hard to combine models from diverse groups of users even when they share conceptual frameworks and software systems. The unnecessary duplication of work is a frequent result. It is very hard to combine models from dissimilar systems. Our hypothesis is that we can overcome many of these tool sharing and reuse problems and significantly improve future model building.

All the problems we have described are compounded as projects grow larger. Many modeling methodologies and software tools have been proposed to address the needs of one or another particular kind of specialist, but very few have been shown to be effective on very large problems solved by teams of specialists making diverse contributions. One technology that has been successfully scaled up to tackle very large industrial problems is the modeling of physical and economic systems with equations. Specialists in equation solving contribute formulation and solution algorithms. Specialists in modeling some particular kind of physics contribute model equations in explicit form (equation-based modeling languages) or implicit form (matrices derived from discretized partial differential equations). These equations can be reused in solving other problems so long as the assumptions required to write the equations still hold. The resulting models in principle may be used for simulation, optimization, parameter estimation or any computation other than that for which the models were originally created since the equations are independent of the solution procedures. Depending on the particular physics and software involved, equation systems containing up to $1\text{x}10^9$ variables can currently be

2. Consider, for example, the similarity of spreadsheets, process simulators, and partial differential equation modeling tools. All are used to create and solve mathematical problems, but there the similarity ends. The assumptions and computations underlying each are rather different.

solved. Creating equation-based models of such complexity, particularly models of never-before-considered chemical processes, is still very difficult. We must combine the work of many modelers to create these models efficiently.

1.2 OUR SOLUTION

Our solution to handling the six problems noted in model reuse is to propose an open conceptual framework based on the following components.

- Open hierarchical representation.
- Dynamic configuration within set model structures.
- Methods and documentation bound to models.
- Generalized equation-based modeling.

We require two abilities to address problem five which is an issue in managing information of all kinds.

- Tracking changes in variable model data and examples of model application.
- Locating appropriate models and tools as they are needed.

Open, hierarchical representations allow different views of the same information. They allow replacement of independent submodels with improved submodels during the evolution of the overall problem. They allow the division of modeling labor. They reduce the information that must be viewed simultaneously by the user. They can expose the information required to make each level of a complex model well defined, if they are constructed in a manner we will suggest. When problems occur in the physical system being modeled, we may suddenly need to scrutinize intensely any part of the corresponding models. All levels of detail within the total system model hierarchy must be open to our examination since we can seldom predict which parts of the system will present problems or interact in unexpected ways. In Section 3.3 we propose design goals and features for modeling components that could be used to build up open, hierarchical representations. In Section 5.5 we show how we have realized an open, hierarchical representation with the ASCEND IV language.

Dynamic configuration means dynamically controlling which one of several existing submodel alternatives is active based on discrete problem variables. All the alternatives are constructed as parts of the larger model structure, but only the active subset of the models is solved. This provides a mechanism for modelers to turn off conflicting parts of submodels selectively *provided the modelers can see the key features of the submodels and understand what they see*. With this mechanism available, it is possible to resolve conflicts in the models and assumptions made by previous modelers. Algorithms capable of solving for both discrete and continuous variables could be used to determine which submodels are active if each alternative represents one of many valid possibilities, for example, in heat exchanger network synthesis techniques based on network superstructures [1]. We give an example where conflict resolution is necessary in Section 3.3.1. In Section 5.2 we show that our conflict resolution mechanism can also be used for more general conditional modeling.

Methods and documentation bound in computable form to models and submodels in a model hierarchy can capture and spread current modeling knowledge. Users can access and reuse expert knowledge coded in methods to perform common model manipulations such as configuration, initialization, scaling, loading or saving data, solving, and solution analysis. In an open system, methods can also call methods of foreign objects to perform tasks not implemented directly in the system. Documentation bound to a model (in contrast to comments that are discarded from the source code when a model is loaded) can be used to communicate possible model applications to unfamiliar users. It can also supply information needed to forge connections to other tools, such as other modeling systems, graphic user interfaces, and tools that search over hierarchical structures. Documentation bound to a model is more likely to be accurate than documentation which exists independently and is updated less frequently than the model. In Section 5.4 we present examples of our computable documentation in ASCEND IV.

Generalizing equation-based modeling captures more general relationships than current equation-based systems. It can capture the logical, linear, nonlinear, and possibly statistical constraints common to models of physical and economic systems. It can also capture the structural relationships (part/whole, superclass/subclass, and similarity relationships) among equations and models. It supports reuse of models by not permanently hiding details which may become important in unanticipated applications. It requires sharing more than just the equation residual and gradient information of a model. It requires a more open view of what modeling systems and solution tools must be since there is no universally appropriate automatically computable mathematical modeling language. We shall present the concepts underlying our extensible solver software architecture for manipulating and solving generalized equation-based models in Section 3.3.

Tracking changes in variable model data is a general information management problem not limited to mathematical modeling. Finding existing models and tools so that users do not waste time reinventing solutions is also a general problem. By defining an open framework for mathematical modeling, we allow ourselves to reuse the tools and techniques being created by information management specialists. Thus, we save ourselves from reinventing information management tools, and we are free to focus on improving the methods and tools of our own field: modeling systems to support engineers.

By defining an open framework for mathematical modeling, we also allow others to reuse our tools and techniques. Thus the ultimate responsibility for controlling our reusable tools must be assumed by the end users of our tools or by software agents that they employ. We cannot assume that our mathematical modeling framework or any other particular modeling tool defines the complete universe of the end users, because that assumption denies the reality that new tools, models, and viewpoints are constantly being created to improve the problem solving process. It is not enough for us to throw new pieces (frameworks and tools) into the modeling puzzle randomly; we must also suggest how the pieces fit in the overall picture and what that picture could look like. We discuss our picture further in Chapter 2 and Chapter 3.

1.3 OPEN MODELING ISSUES

There are four modeling issues we must highlight because they have a particularly large impact on the details of the conceptual framework presented in this thesis. The issues are not unique to open mathematical modeling systems or to process engineering systems, but we must understand them in order to construct a framework that allows us to reuse the work of many modelers.

1.3.1 SHARING COMMON INFORMATION

When creating complex physical models there is usually information shared between models, for example, a stream is common to the unit which is its source and the unit which is its destination. We may wish to solve either unit model alone, possibly making unit specifications which require computing the stream properties. Both source and destination unit must have the complete model of the stream if we are to use the units in this fashion; one unit cannot be responsible for computing the stream properties for the other.

There are at least two ways to handle this situation systematically without making restrictive assumptions about pair-wise model connectivity that are peculiar to chemical engineering. One way is to define a stream in each unit and then declare a merge [5] of the stream from one unit and the corresponding stream in the other unit as part of the flowsheet model definition. We show this in Figure 1-1. The first difficulty in this merging is that we *must* read and understand in detail the code describing the units in order to find the parts named “out” and “in” that we need to merge. The second difficulty is that merge operations can slow the computations needed to construct a working model object by a factor of two to ten, as we shall see in Section 5.3. Another way is to create the stream in the flowsheet model and pass the stream object to the two units that share it through the interfaces defined for the units. We show this in Figure 1-2.

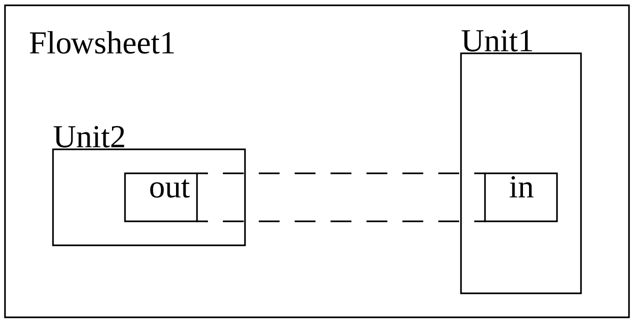


Figure 1-1 Merging stream Unit2.out with Unit1.in for Flowsheet1

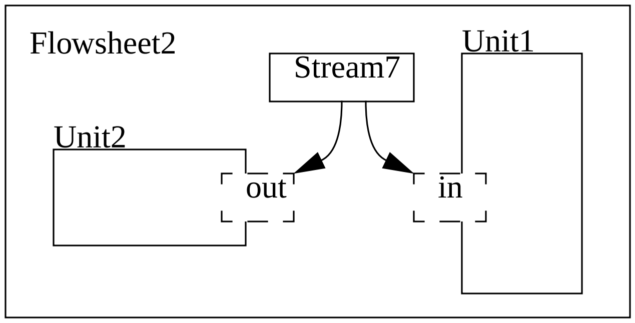


Figure 1-2 Passing a common stream to both units through interfaces

If we model in the style of Figure 1-2, then connections between models become obvious, and it easier for us to modify the connections when we must modify the flowsheet. We also gain by eliminating the wasted computations required to construct two streams before merging them into one. Finally, by creating a single object and passing it to all the locations that need it, we ensure that all submodels are working with consistent data. For example, we could define an object containing the physical property options and a list of chemical species that are to be used throughout the flowsheet and pass this object to all streams and units in the flowsheet. Our realization of the object passing concept is described in Section 5.5.

1.3.2 ANONYMOUS TYPES

In the world of computational modeling, a type (a class) is a set of statements which describes a group of similar objects. Processing the statements of a type to construct an object is called instantiation or compiling. In mathematical modeling, the set of relations and the other statements needed to construct them constitutes a type. There are two ways in which a model type may be reusable. First, it may be a complete set of statements containing all the information needed to instantiate a mathematical object using a given algorithm. In modeling physical systems, thousands of complete types may be available to choose from, as is the case with simple parts such as resistors, capacitors, and integrated circuit components in an electronic parts catalog. Second, a type may be an incomplete set of statements which leaves detailed specification of certain features within the mathematical object to the end user. For example, a reusable rigorous distillation column model may leave the choice of chemical species, of number of trays, and of thermodynamic calculation methods to the user; this column type is *incomplete*. The type of the column object compiled from the incomplete column type and the final user specifications is *anonymous* because nowhere has anyone written down a single complete set of statements that describe the resulting mathematical object.

Most mathematical objects used by chemical engineers have anonymous types because creating an explicit complete type for every combination of final specifications leads to extreme difficulty in storing, locating and reusing models. For example, consider the number of complete types we can define for a simple distillation column where we have 25 thermodynamic calculation method alternatives, a number of trays between 3 and 100, and mixtures combining five chemical species from a set of 1500. Both creating and searching a catalog of over 10^{15} such distillation models would be ridiculously expensive.

One way we can evolve a structured initial process model object (already full of anonymous types determined by the species, number of stages, and so forth) to a more detailed model is to reach inside the structure and extend the simple thermodynamic calculation (perhaps using constant relative volatilities) to a more refined type (a subclass) of thermodynamic calculation. If we can model this way, then we can more quickly create and explore organizations of types (model libraries) when the best organization is not yet clear. This model evolution method *requires* creating anonymous types.

Unfortunately, models that are created by merging parts of submodels or by reaching inside submodels to make anonymous type changes have proven quite difficult to reuse, as we discuss in Section 4.3.2. We need ways to take our discoveries from early exploratory models and restate them explicitly in models which are easy to reuse. Consider a simple distillation column which uses the same simple thermodynamic model on each tray; the anonymous types of all the trays except the condenser, feed tray, and reboiler are similar[3]. If we refine the thermodynamic calculation on each tray using a distinct thermodynamic

3. The condenser, reboiler, and feed tray have distinct anonymous types because they contain information about input and output streams which the simpler stage models do not. The condenser, reboiler, and feed tray may even be constructed from explicitly different model types, depending on the modeling system and modeling style in use.

subclass, then each tray in the column has a distinct anonymous type. It is quite probable that the column mathematics require the same thermodynamic model to be used on all the trays, so we need mechanisms which allow us to define and refine at the same modeling level both the column and the thermodynamic calculations which will be used. If we can do this, then we can also define mechanisms which prohibit changes in the anonymous type of an object unless those changes are made at the proper modeling level. These mechanisms enable us to prevent incorrect use of open model structures and to make model types more understandable when modifications are required. We will show in detail how we can achieve this in Chapter 5.

1.3.3 OPEN OBJECTS AND "MODELING BY CONTRACT"

One of the central ideas in "object-oriented (OO) programming" is information hiding. Traditional OO programming objects (data and associated imperative code) share information by sending messages through message passing interfaces so that the data and implementation details may be hidden. Because we need access to practically all of the data in our model objects in order to resolve modeling conflicts and to handle other problems encountered in applying the models, information hiding at the mathematical modeling level is inappropriate and message passing simply adds unnecessary computational overhead.

While rejecting information hiding at the modeling level, we can still learn much from the continuing work of computer scientists on object-oriented design methodology which can substantially improve the reusability of our models. Even though a mathematical model is "open," meaning we can see all its variables, equations, and submodels down to the last detail when necessary, we must be able to use the model in routine applications without needing to understand all these details if we are to construct large models efficiently. A "design by contract" methodology, which might be more accurately called "programming by contract," is proposed by object-oriented software design researchers to support software reuse [2,4]. A practical implementation based on this methodology, Eiffel [3], is reported to remedy the failures of "portable assembly" languages such as C++ and FORTRAN to support efficient software reuse in very large software systems. We can draw parallels to "programming by contract" in an open system of mathematical objects to define “modeling by contract.”

The essential feature of software design by contract is that it explicitly associates with each software object an interface and computable statements defining:

- the preconditions which must be met to use the object.
- the class invariants, relationships about objects and object data which always hold for any computation performed by that object.
- the postconditions, relationships which the object guarantees to satisfy when it is used correctly.
- the exception handling protocols required when an object is incorrectly used.

In our equation-based modeling method, types (classes) can be easily reused if we too make the preconditions of use explicit. Our types are primarily data structuring tools, so the data (parts) in our open models are our class invariants. Because our open models simply state the existence of equations, variables, and submodels but not algorithms, we do not need to contend with postconditions or exception handling when defining the type structures. In Eiffel, the precondition statements have relatively simple forms, checking relationships among types and values of input parameters. If we are to support modeling with open structures and object passing, however, we must extend the concept of preconditions to capture easily restrictions on deeper structural relationships as well. We will show how these relationships can be stated and checked in Chapter 5. Using examples of chemical engineering models, we will see that these relationships are often rather complex, which may in part explain the difficulties we currently encounter in reusing poorly documented engineering modeling software.

1.3.4 INTELLECTUAL PROPERTY AND OPENNESS

If we define completely open model structures to facilitate model reuse and abandon model information hiding, we have eliminated one layer of protection for the intellectual property rights of model authors. Unlike a computer program compiled into binary form, a completely open, reusable computer model is easily recreated in different forms. For modeling within a single enterprise, this does not present a difficulty as the intellectual property rights of all the employees are usually ceded to the enterprise. However, third party suppliers of modeling technologies may find little motivation to supply open models if these suppliers cannot find a way to extract sufficient profit from the customers before the information contained in these models diffuses into the store of common engineering knowledge. We discuss this situation further in Section 2.2.

Our contributions to the concepts of reusable modeling may be of little economic value until the creators and users of models agree on a definition of fair exchange that can accommodate open sharing of information. The precise economic worth of our ideas is only mildly an academic concern, but we must not fail to note the inadequacy in the present concepts of intellectual property which could discourage further investigation of our ideas by others. The challenge to scholars in areas of information science and property law is to find a systematic way to encourage new developments in a marketplace of ideas where the full details of each idea are made visible to the buyer.

1.3.5 ATTACKING THE PUZZLE

The problems of model reuse, our solutions, and the issues which attend both are parts of the larger picture of systems research. We need to show where our solutions fit in the overall picture so their merits can be determined by others. We begin this display in Chapter 2 with a look at the roles of modeling in design and of equation-based modeling in the modeling systems landscape. In Chapter 3 we consider the traditional object-oriented, information-hiding approach to modeling software, and we present our contrasting view of a more reusable mathematical modeling software component. In Chapter 4 we consider how not one but many languages must be combined in an open modeling environment, and we suggest desirable properties for any language which is to play a role in such an environment. In Chapter 5 we present the design rationale for and some examples of ASCEND IV, a language we are implementing to test our concepts of open modeling languages and open modeling environments. In Chapter 5 we also examine two language concepts apparently new to chemical engineering, object passing and modeling by contract, and we then compare the language we have defined in Chapter 5 against the desirable language properties listed in Chapter 4 to see where other tools and concepts may be needed. In Chapter 6 we describe several new tools that help the user manipulate information in complex models, and we propose future work that would further enhance model reuse.

[1] L. T. Biegler, I. E. Grossmann, and A. W. Westerberg. *Systematic methods of chemical process design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1997.

[2] Bertrand Meyer. “Applying ’design by contract’.” *IEEE Computer*, pages 40–51, October 1992.

[3] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[4] Jean-Marc Nerson. “Applying object-oriented analysis and design.” *Communications of the ACM*, 35(9):63–74, 1992.

[5] Peter Piela, Roy McKelvey, and Arthur Westerberg. “An introduction to the ASCEND modeling system: its language and interactive environment.” *J. Manage. Inf. Syst.*, 9(3):91–121, Winter 1992-1993.

CHAPTER 2 CHANGING MODELING SYSTEMS

2.1 DESIGN

The design of complex engineered systems is a difficult and error-prone social process [4,15]. Engineering design has recently become even more difficult as economic and regulatory pressures to account for the entire product life-cycle in all phases of design have sharply increased. Various structured modeling methods have always been used in chemical engineering. Indeed, some believe the view of chemical processes as structures composed of unit operation models is what first differentiated chemical engineering from other engineering disciplines [13]. The working environment of an engineer is itself a complex engineered system, and it is very hard to create such an environment which gives the engineer fine control of the ever-growing array of tools and kinds of information necessary to handle both routine and non-routine tasks effectively. The work environment should constitute a stable but easily expanded base which supports the engineer and the overall goals of the business enterprise [45].

The structured process modeling habits of chemical engineers have naturally been reflected in the design of computer systems for process modeling created by chemical engineers. Each commercial simulator has a proprietary, handcrafted library of commonly used unit operation models and a graphic user interface (GUI) to hide and manage the details of applying that library. The striking similarities between configuring a set of unit operation models to create a flowsheet and configuring a set of computational objects to create arbitrary software [29] has lead some to an interesting idea [17] which can be summarized as follows.

> The overall task of process modeling could be much better done if unit operations were packaged as standardized components. We could use any off-the-shelf total chemical process modeling environment we prefer to create and manipulate process models out of parts from many different proprietary libraries.

A frequently proposed example of this idea is to take one vendor's thermodynamic models, another's distillation models, a third's heat exchanger models, and combine all these standardized models with an in-house reactor model to obtain an overall flowsheet.

We take a hard look at this idea and the landscape it inhabits in this chapter. This thesis will not answer all the issues we raise. Instead we aim to present possible solution paths for some of the issues which make us believe that overall the problems are not insurmountable. Others have argued that designing standardized interfaces for streams, physical properties, and common unit operations is a worthwhile step in the evolution of chemical engineering software systems [11]. We agree, but we contend it can be only an early step if we aim to support the engineer and the business effectively.

We see the first difficulty in the assumption that an off-the-shelf modeling environment designed primarily around a chemical engineer's analysis, simulation, and data management needs is also the best solution to a chemical manufacturer's information and organization management problems. This supposition seems tenuous at best, given studies [18,44] which suggest that only about 15% of an engineer's time is spent in analysis related tasks. A better supposition might be that engineers and other technical contributors in the overall business enterprise should rapidly and inexpensively build modeling environments suited to the domain specific (or interdisciplinary!) modeling tasks at hand from a range of pre-packaged but extensible software components.

We would build such an ideal modeling environment based upon internationally standardized interfaces [2], managing the software components through a general information modeling tool more appropriate to the overall business enterprise such as *n*-dim [45]. In our ideal world, each of the features incorporated in today's gigantic, off-the-shelf modeling environments could be repackaged as a component. The eventual result of such a shift in modeling practice would be to put the manufacturing enterprise back in control of its modeling technology in an inexpensive way, a result which seems to be the opposite of current trends [8,41] in chemical process modeling software.

The second difficulty we see in the widespread component view of software is its *incompatibility with reusable 'open form modeling'*. Open form modeling is the creation and then simultaneous solution of a set of equations describing process physics [22], and it has been identified by many researchers (both commercial and academic [22,41,31]) as a key element in the delivery of large-scale (plant-wide or enterprise-wide) simulation and optimization technology.

We finish this chapter with a review of why open form modeling is necessary and why, in general terms, we think it has incompatibilities with the component software view. In the remainder of the thesis, we will contribute an analysis of open form modeling information needs and possible solution elements to several key areas in the very difficult task of designing open form modeling system standards. We intend to make clear that open form models could indeed be standardized and that a one to two orders of magnitude decrease in the present cost of building truly reusable modeling systems is possible.

2.2 OPEN FORM MODELING PROBLEMS

Open form modeling is more commonly known as 'equation-based modeling.' The term "open form" has been adopted recently [22], possibly to emphasize the point that if one combines unit operation models which are internally equation-based but each model hides those equations from the flowsheet solver, the overall efficiency of an "equation-based" solution method does not appear. Equation-based modeling advocates [16] reached the conclusion more than two decades ago that solving all the process model equations simultaneously is frequently a much faster approach than iterating through a coupled sequence of equation subsets. Such an iteration scheme is known as the sequential-modular approach, and it remains the standard industrial practice for most chemical process simulations because it often avoids convergence difficulties encountered in Newton methods with poor initial solution estimates [22]. The modular approach can cause convergence difficulties that would not appear in an equivalent simultaneous problem [16], a situation not as uncommon as most simulation and optimization users would like it to be.

The solution of open form models has been demonstrated as effective in solving the simulation, optimization, parameter estimation and data reconciliation problems [22,41] all using a single set of equations. A different solution algorithm may be used for each problem, and each problem is characterized by which variables in the total model are held fixed during the solution process. The use of such multi-purpose models throughout the process plant life-cycle has been proposed by several authors [32,23,7] and is today the foundation of advanced real-time optimization and control systems.

There are many unresolved issues in the creation and use of open form modeling systems.

- It is considered very difficult to create general purpose software environments which lead effortlessly to well-posed models [22].
- Nonlinear equation-based models, even when well-posed and object-oriented, are generally described as very hard to use [22] and are often seen as requiring a paradigm shift [9] from thinking about unit operations to thinking about equations.
- The examples of general equation-based modeling tools published so far do not scale up to very large problems except under restrictive assumptions [6,5,10,24].
- Recreating an open form model written in any equation-oriented computer language is relatively easy given another equation-oriented computer language, as we shall see. [25,26,12,10,7]

The difficulties of scaling general equation-based modeling tools to very large problems have been described in [6]. The scalable systems so far described in the literature obtain scalability by sacrificing ease of use [6] or generality [5,10,24]. We wish to challenge the assumption that a scalable system will be either specialized or hard to use. We suggest that our efficiency in posing very large mathematical models and understanding the results of them is strongly influenced by the tools and language we use. One of our goals in this thesis is to develop a modeling language which is efficient from a user's viewpoint, a language that makes the relationships in complex model structures clear. We shall see in Chapter 5 that several ideas which help to clarify models for users also provide information which can be used to improve the efficiency of modeling computations.

We think the fourth point about recreating models makes the component software view incompatible with open form modeling environments. That is, recreating a similar model in an equation-based language or a strongly object-oriented imperative language is possible with modern computer-aided software engineering (CASE) tools [25,26] or equation-based modeling systems [12,10,7]. For example, we can mechanically translate a compiled ASCEND model into GAMS code, thus we only need to use ASCEND to create the model. We can make routine use of the model without running ASCEND, thus we can avoid any fees associated with ASCEND[1]. The licensors of open form models must obtain legal protection of the original models or lower the costs for production and use of such models to the point where piracy by model recreation is not worth the licensee's effort. A second alternative for protecting open form models is to create open form models in a language which makes the model difficult to replicate (perforce, difficult to use) without licensing the executive system which provides friendly interfaces to such models. A third alternative is for the vendor to provide 'black box' model interfaces to the user, where these interfaces merely contact the vendor's computers across a network to retrieve standard information such as equation residuals and gradients based on the user's input. If the definition of "standard information" is too narrow, this last alternative might not be a significant improvement over current commercial simulators.

2.3 THE MODELING SYSTEMS LANDSCAPE

Before suggesting partial solutions to the open form modeling problems, we should first examine the more general landscape that open form technology must inhabit. We survey three key features in this landscape.

First, highly sophisticated, mature CASE tools are now available and getting better all the

1. There are no fees associated with ASCEND, and we do not anticipate that there ever will be. This is not the case with all modeling systems, however.

time, making the production of sophisticated, reliable, low cost software components in a short amount of time possible, or so some would have us believe [25,27,38,29]. The thousands of hours once spent looking for memory management errors in low level languages such as C++ and FORTRAN have been eliminated by truly object-oriented languages [25] which make automatic memory management a viable option. Even when speed critical applications still require the use of low level languages, auditing tools such as Purify [37] make long hours spent on hunting for memory management errors largely a thing of the past. More importantly, object-oriented programming methods and systems aimed at solving the many inter-organizational and information management issues surrounding data exchange and the general *software design* problem are now sufficiently well-developed that international standards [2] are being proposed and adopted in many application areas including process engineering [1,3]. These object-oriented CASE tools and methodologies share the assumption that it is easier to implement and reuse software if the details of data structure and implementation of a given object are maximally hidden. We shall suggest in Chapter 3 that easily reusable mathematical modeling requires that very detailed model information must be available. The essential tension between the CASE concept of information hiding and the open form modeling requirement of wide open data communications may make open form modeling component software standards very hard to develop.

Second, people from many disciplines and subdisciplines, each with distinct technical languages and methodologies, must contribute knowledge to the overall task of modeling a plant and the associated business processes during the chemical process life-cycle [14,39]. Frequently the overlaps and subtle distinctions in terminology make collaboration in model building difficult [14]. One approach taken by several research groups [28,10,23,47] and at least one process modeling software vendor [22] has been to suggest that the various disciplines and degrees of user sophistication can be handled by multiple specialized graphic user interfaces (GUI) layered over a single modeling language. The underlying languages proposed are commonly object-oriented, equation-based, and specified using text file input. They are designed primarily with well trained users or experts in mind. Reported input languages taking this approach have not addressed their incorporation as components into other equally sophisticated modeling systems which have radically different modeling representations. For example, none of the research environments mentioned [28,10,23,47] propose a general method for sharing information with finite-element models (FEMs) defined on irregular three-dimensional meshes in a way that results in the equations generated in each environment being combined for simultaneous solution.

We, among others, have a natural tendency to recast the highly evolved methods and models of other domains in our own domain specific terms [43]. Important interdisciplinary connections are often discovered in this way, but in many cases we wonder if the overall goal of life-cycle modeling might not be better served by concentrating our efforts on making the tools of our own domain better. We must respect and utilize the contributions from the best researchers and commercial software developers in all disciplines if we are to meet the overall goal of producing less expensive, higher quality tools to support the overall chemical processing enterprise. The practical implication is that we need to accommodate the specialist languages of each domain in an overall software environment designed with multiple languages and multiple users in mind. We should construct this environment according to the best available software engineering standards. There are specialists in the computer supported collaborative work area dedicated to creating such environments [21,42].

Third and finally, in spite of the supposed inefficiencies of the sequential unit operations modeling approach (it can be both mentally [40] and computationally inefficient), this approach and the software that supports it must be accommodated during a period of model migration which will last at least until the advocates of equation-based modeling have demonstrated the clear superiority of their methods for all situations. In fact, sequential-modular software is being used by one of the strongest advocates of open form modeling [22] to initialize their open form models, thus avoiding the expense of recreating the time tested initialization methods embedded in the sequential-modular models. Instead of eagerly awaiting the retirement of the sequential-modular modeling paradigm, *we should respect and utilize the knowledge available in “legacy” codes where these codes have proved particularly effective*.

2.4 OPEN FORM MODELING SOLUTIONS

ASCEND III [36,6] and similar equation-based general purpose modeling systems [28] are not apparently receiving wide industrial use. The object-oriented mathematical modeling paradigm as so far demonstrated by university examples has apparently been perceived as one or more of:

- Requiring such an enormous shift in user thinking as to be an infeasible alternative at present for anyone but the most expert modelers [9].
- Unscalable to realistic problem sizes [6].
- Not adequately addressing the really hard issues in process modeling, which are more organizational than mathematical as seen in the two previous sections [45].

The first point has been acknowledged as a significant problem by many authors, [31,24,28,36,22] but the academic codes continue to be designed by and for expert modelers. The difficulty cannot be object-oriented thinking because, as noted in Section 2.1, chemical engineers have a long tradition of object-oriented methodologies. At least one group [9] has questioned whether or not dealing with equations will prove beyond the abilities or at least the time constraints of most modeling users. The problems associated with equation-based systems include formulation, initialization, scaling, convergence, and interpretation of the results. Work on each of these problems has been ongoing in many fields [20,30,46].

The second point, however, must be addressed in order for research on very large scale problem solvers and modeling systems to advance. Currently, interactive object-oriented, general purpose systems do well to create and manage 20,000 to 50,000 equations on a workstation with 200 megabytes of fast memory and a 150 megahertz RISC processor [6]. The behavior of very large systems (100,000 to 1,000,000 general form nonlinear equations) cannot be very well investigated if practical, cheap, usable, and, most importantly, modifiable software tools for creating and testing such large models are not available.

The first point has been acknowledged as a significant problem by many authors, [31,24,28,36,22] but the academic codes continue to be designed by and for expert modelers. The difficulty cannot be object-oriented thinking because, as noted in

Section 2.1, chemical engineers have a long tradition of object-oriented methodologies. At least one group [9] has questioned whether or not dealing with equations will prove beyond the abilities or at least the time constraints of most modeling users. The problems associated with equation-based systems include formulation, initialization, scaling, convergence, and interpretation of the results. Work on each of these problems has been ongoing in many fields [20,30,46].

The university codes cannot be faulted for neglecting the third point because they have been designed primarily with other research goals in mind, such as exploring mathematical or user interface issues in dynamic process simulation [10,7] or optimization. Other research systems are being created to address organizational issues in modeling [45].

We may obtain insights that help address the first two points by examining the existing modeling systems in the light of several years of accumulated user experiences. We need object-oriented mathematical modeling tools that give conventional simulation users a gentle migration path to more advanced modeling techniques in a system that by design includes model verifiability and model scalability as two of its primary properties. We will define such a system, contrasting it most frequently with our own ASCEND III [35,19,34] and ASCEND IIIc [6,33] systems since it is primarily these experiments which lead us to the conclusions that motivate the construction of ASCEND IV. We will make comparisons with the claims of other systems where it is practical and relevant, and we will observe some contrasts with the component software view of the future in process modeling.

Based on the discussion in this chapter, we suggest at least three changes that are required to deliver equation-based mathematical modeling technologies in support of the overall business enterprise:

- Adoption of an expanded view of modeling system structures.
- Creation of better languages for mathematical modeling as part of a larger enterprise.
- Adaptation of existing and creation of new supporting algorithms and tools for open form modeling.

Each of these changes will be discussed with examples. The major contributions of this work are in the areas of open form modeling system design and language design insights. Our test implementation of the modeling language proposed is largely complete and early experimental results are presented in Chapter 5.

[1] “ISO 10303 application protocol 221: Process plant functional data and its schematic representation.” International Standards Organization.

[2] “ISO 10303 standard (STEP).” International Standards Organization. Standard for the Exchange of Product Model Data.

[3] “ISO application protocol 231: Process engineering data.” International Standards Organization.

[4] “Improving engineering design: Designing for competitive advantage.” National Academy Press, Washington D. C., 1991. National Research Council report.

[5] *SpeedUp User Manual*. Cambridge, Massachusetts, 1991. Aspen Technology.

[6] Kirk Andre Abbott. *Very Large Scale Modeling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1996.

[7] Mats Andersson. *OMOLA - An Object-Oriented Modelling Language*. PhD thesis, Lund Institute of Technology, Department of Automatic Control, Lund, Sweden, 1990.

[8] Aspen Technology. *ASPENWORLD 94*, Boston, MA USA, November 1994.

[9] William L. Barnard, Dean R. Benjamin, Daniel L. Cummings, Peter C. Piela, and James L. Sills. “Three issues in the design of an equation-based process simulator.” Presented at the AIChE Spring National Meeting, Paper 42d, 1986.

[10] Paul I. Barton. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, 1992.

[11] N. O. Book, R. Motard, M. Blaha, B. Goldstein, J. Hendrick, and J. Fielding. “The road to a common byte.” *Chemical Engineering*, September 1994.

[12] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user’s guide*. Scientific Press, 1988.

[13] George Brown. *Unit Operations*. Wiley, New York, 1950.

[14] L. L. Bucciarelli. *Designing Engineers*. MIT Press, 1995.

[15] George Dieter. *Engineering Design*. McGraw-Hill, New York, 1983.

[16] F. C. Edie and A. W. Westerberg. “A potpourri of convergence and tearing.” *Chemical Engineering Computing*, 1:35–39, 1972.

[17] Peter D. Edwards and Gary Merkel. “Impact of an open architecture environment on the design of software components for process modeling.” In James F. Davis, George Stephanopoulos, and Venkat Venkatasubramanian, editors, *Intelligent Systems in Process Engineering: Proceedings of the Conference held at Snowmass, CO July 1995*, volume 92 of *AIChE symposium series*, pages 307–310. CACHE, 1996.

[18] R. S. Engelmor and J. M Tenenbaum. “The engineer’s associate:.” ISAT summer study report, 1990.

[19] Thomas Guthrie Epperly. “Implementation of an ASCEND interpreter.” Technical report, Engineering Design Research Center, Carnegie Mellon University, 1989.

[20] Harvey Greenberg. “Enhancements of analyze: A computer-assisted analysis system for mathematical programming models and solutions.” *ACM Transactions on Mathematical Software*, 19(2):233–256, 1993.

[21] S. Levy, E. Subrahmanian, S. L. Konda, R. F. Coyne, A. W. Westerberg, and Y. Reich. “An overview of the n-dim environment.” Technical Report TR 05-65-93, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA USA, 1993.

[22] Vladimir Mahalec. “Software architecture for on-line modeling and optimization.” Presented at AIChE Spring National Meeting, Houston, TX USA, 1993. See also: http://www.aspentech.com.

[23] W. Marquardt. “Trends in computer-aided process modeling.” *Computers and Chemical Engineering*, 20(6/7):591–609, 1996.

[24] Wolfgang Marquardt. “An object-oriented representation of structured process models.” *Computers and Chemical Engineering*, 16S:S329–S336, 1992.

[25] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[26] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[27] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. “Component-oriented software development.” *Communications of the ACM*, 35(9):160–165, September 1992.

[28] Bernt Nilsson. *Object-Oriented Modeling of Chemical Processes*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[29] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, September 1995.

[30] Jorge R. Paloschi. “Bounded homotopies to solve systems of sparse algebraic nonlinear equations.” *Computers and Chemical Engineering*, Accepted 1996.

[31] C. C. Pantelides and P. I. Barton. “Equation-oriented dynamic simulation: current status and future perspectives.” *Computers chem. Engng.*, 17S:263–285, 1993.

[32] C. C. Pantelides and H. I. Britt. “Multipurpose processs modeling environments.” In L. T. Biegler and M. F. Doherty, editors, *Proc. Conf. on Foundations of Computer-Aided Process Design 94*, CACHE Publications, pages 128–141, 1995.

[33] Jennifer L. Perry and Benjamin A. Allan. “Design and use of dynamic mod-

eling in ASCEND IV." Technical Report EDRC 06-224-96, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1996.

[34] Peter Piela, Thomas Epperly, Karl Westerberg, and Arthur Westerberg. "An object-oriented computer environment for modeling and analysis: The modeling language." *Computers and Chemical Engineering*, 15(1):53–72, 1991.

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

[36] Peter Colin Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[37] PureAtria. http://www.pureatria.com, Sunnyvale, CA USA. *Purify User's Guide*, 1994.

[38] PureAtria. http://www.pureatria.com, Sunnyvale, CA USA. *Quantify User's Guide*, 1994.

[39] Jerry L. Robertson, Eswaran Subrahmanian, Mark E. Thomas, and Arthur W. Westerberg. "Management of the design process: the impact of information modeling." In Lorenz T. Biegler and Michael F. Doherty, editors, *Fourth International Conference on Foundations of Computer-Aided Process Design*, volume 91 of *304*, pages 154–165, Snowmass, CO USA, 1994. CACHE.

[40] Dale F. Rudd, Gary J. Powers, and Jeffrey Siirola. *Process Synthesis*. Prentice-Hall, 1973.

[41] Simulation Sciences. "Rigorous on-line modeling." http://www.simsci.com, March 1997. Product literature indicates that Simsci will do your modeling for you, selling a total process solution.

[42] C. Simone, M. Divitini, and K. Schmidt. "A notation for malleable and interable coordination mechanisms for CSCW systems." In *Proceedings of the Conference on Organizational Computings Systems (COOCS)*, pages 44–54, New York, 1995. ACM Press.

[43] E. Subrahmanian, S. L. Konda, S. Levy, I. Monarch, Y. Reich, and A. W. Westerberg. "Computational support for shared memory in design." In A. Tzonis and I. White, editors, *Automation Based Creative Design: Current Issues in Computers And Architecture*, Amsterdam, The Netherlands, 1993. Elsevier Publishers.

[44] E. Subrahmanian, S. L. Konda, S. N. Levy, Y. Reich, A. W. Westerberg, and I. A. Monarch. "Equations aren't enough: Informal modeling in design." *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, 7(4):257–274, 1993.

[45] Eswaran Subrahmanian, Yoram Reich, Suresh L. Konda, Allen Dutoit, Douglas Cunningham, Robert Patrick, Mark Thomas, and Arthur W. Westerberg.

"The n-dim approach to building design support systems." *Submitted to ASME Design Theory and Methodology*, 1997.

[46] Kenneth Tyner, Benjamin Allan, and Arthur Westerberg. "Scaling nonlinear equations." Technical Report EDRC/ICES TR 0-0-0, Engineering Design Research Center, Carnegie Mellon University, 1997. In preparation.

[47] Arthur W. Westerberg and Dean R. Benjamin. "Thoughts on a future equation-oriented flowsheeting system." *Computers and Chemical Engineering*, 9(5):517–526, 1985.

CHAPTER 3 AN EXPANDED VIEW OF COMPONENT-BASED MODELING

For a manufacturing company of any kind, engineering of all kinds (including mathematical modeling), is simply a necessary expense. In large enterprises such as petroleum processing where raw material purchasing strategies and product prices dominate the financial balance sheets, engineering and process optimization may not even be a dominant factor in profitability. Whether to purchase engineering services from direct employees or by contracts with outside consultants is a matter of which service source provides the cheapest method of getting immediate engineering tasks performed without incurring an unacceptable risk in terms of future costs. Future costs could come from damage to equipment, to employees, or to the environment due to insufficient design or from extra engineering work required to recreate past information which is subsequently lost. Component software advocates share the view that knowledge, as embodied in computer software, could be more effectively and less expensively purchased if it were available in precisely targeted pieces rather than available only in a complete body [13].

In Section 3.1 we characterize component software success and failure modes. In Section 3.2 we present a view of how chemical process modeling might be aided in the future by adopting a rigorously component-oriented view of the software tools, and we discuss some of the general implications for researchers and software vendors. In Section 3.3 we present our view of what is required to make an open form modeling component reusable.

3.1 A PERSPECTIVE ON CURRENT SOFTWARE

Component software provides a means to put individual tools into the hands of users as the specific tools are needed. The successes demonstrated (or at least promised) by examples of component software in areas such as graphic processing and construction of numerical algorithms should be examined to determine some of the conditions for success or failure of the component software idea. We base the following examples of when component software does and does not work well on experiences gained while creating ASCEND IIIc and ASCEND IV, two mathematical software systems with multiple graphic interfaces. The components that have been considered while building the ASCEND systems include: C, FORTRAN, and PASCAL compilers, compiler tools such as YACC and flex, sparse and dense linear algebra libraries, nonlinear solvers, window building widget sets and scripting languages such as Java and Tcl/Tk, spreadsheet tools, graph plotting tools, World Wide Web browsers, a variety of other software engineering utilities, and, of course, system users.

Software components work when we use them in a single standardized environment. Components designed for one environment almost never work in another. We can inexpensively create components only when using certain major assumptions, for example, when we create numeric applications in ANSI FORTRAN 77 or we create graphic applications for a particular flavor of the Microsoft Windows widget set. Creating component codes that work reliably in multiple operating systems under multiple graphic interfaces using diverse compilers is extremely expensive with current tools and standards because a large quantity of specialized knowledge is needed. Languages such as Java [10] which hide all the computer platform specific details of operating systems, graphic interfaces, and compilers are being developed, but they have not yet evolved to true universality.

Software components work when we can find the right one quickly using easily obtained resources. Each component in our local arsenal should be well-designed for specific tasks, and we should not have to choose a component from among too many candidates. For example, we might want a spreadsheet tool in our modeling environment. If there are about six different spreadsheet-like components available, it should not take us long to determine which one we should use or if none of them is suitable. We probably will reject a spreadsheet which has thousands of expensive features we do not expect to use or one which has precisely the numeric features we need but which is poorly designed for data display. In our selection process we will very likely rely heavily on documentation external to the component objects, such as catalogs, WWW sites, and the opinions of our coworkers based on their experience with similar problems. We may miss a good component choice if any of the documentation sources uses naming and classification schemes that do not match our own domain specific or task specific scheme. We could end up making several expensive purchases before finding the right component.

Once we have purchased them, software components work when we can easily tell if they satisfy our selection criteria. We can easily judge the quality of a component when it gives immediate answers to questions such as: Has the component frozen up my machine? Has my display turned green because the component does not play well with one of my other components? Has the component produced distillation profiles for my test cases which match my benchmarks? Has the component tried to destroy one of the components produced by a competing component provider? We cannot make quick, accurate evaluations of components which do not have such easily evaluated behavior. We need to know somehow that what we see is what we are getting. If we cannot inexpensively verify the performance of a component, we may find it less expensive in the long run to make our own or to accept working with the uncertainties of an unverified component.

Software components work when we can easily understand their overall functionality. This is particularly true when we are finding and using components that are from domains of expertise outside our usual work. If a component has documentation with many different audiences in mind built into it, we are much more likely to find it useful than if it has internal documentation for a single audience of which we are not a member. For example, a process simulator with on-line help oriented only to chemical engineers with less than six months experience can drive more advanced users to frustration in short order.

Software components do not work when we cannot afford them. If we must spend too much time or too much money in order to use a component, it is not a good component. One of the spreadsheet components we considered for adding to ASCEND was technically perfect, but the licensing costs and conditions proved prohibitive.

Software components do not work when the best available is almost what we need and we are prevented from extending it to include the few extra missing features. Components that are available only as executable libraries without extensive, accurate documentation of programming interfaces cannot be wrapped in our own code that adds the missing features. When the features we need require perhaps only small changes to the program logic of a component, we usually cannot add these features at all unless the component source code is available. Software components, and almost any other product, are generally rather better if they are created with strong user input to the design process [20,17].

Software components do not work when we must carry out a trial and error process on too many semantically rich alternatives in the time available to solve the problems at hand. For example, many distinct sparse linear algebra libraries are available for use in creating a nonlinear solver, but we frequently select a locally produced one with properties known to be adequate for the required task rather than risk losing the time required to find and evaluate other linear solvers because we cannot be assured of finding something better in our search.

Software components do not work when we must assemble hundreds of simple components for a single routine task. This can happen when we decide to build a complex object, say a distillation train flowsheet, using only a library with two dozen kinds of state and transition mechanism objects representing various fundamental physical phenomena. We need a collection of components designed with larger granularity, such as unit operations. Each of these coarse grained components might be built from a set of phenomena-based subcomponents, but, as a user, we are not usually concerned with such details.

Software components do not work very well when we cannot see their most important information. What is most important in reusable mathematical models is “everything mathematical.” This includes, but is by no means limited to, the names of variables because the names often give us important clues about how we might alter a variable in order to get a problem to solve. We need the symbolic form of equations because new tools that help in initialization, scaling, decomposition, and reformulation of models by manipulating symbolic information are constantly being developed [14,21,8]. We also need hierarchical structure information from models of physical systems in order to intelligently decompose problems for linear factorization, branch and bound analysis, or algorithms to be executed on multi-processor hardware [3].

Software components do not work very well when we do not have mechanisms for communicating case histories (failures or successes) among users. This is also true of most types of hardware. Government bodies or standards organizations have been created to collect histories, define acceptable practices, and communicate the results as broadly as possible for many kinds of hardware, for example boilers and petroleum storage tanks. Also, components do not work very well when we do not have mechanisms for rapidly evaluating, incorporating, and distributing fixes proposed by the smarter component users. We and others have observed this many times. For example, in a 1995 benchmark of UNIX software components (80 common utility programs) from 7 commercial UNIXes and the freeware Linux, there is an average fatal error rate of 23% among the commercial systems but only 9% for the Linux system [12]. The reasons suggested by the study authors are that some Linux users often take the time to fix bugs and share the fixes, while bug reports (and even potential fixes) sent by users to commercial UNIX vendors are usually not responded to with a new product release in less than a year, if they are responded to at all. Similarly, many of the best features of the ASCEND system interfaces originated from the suggestions of frustrated, and not necessarily advanced, users rather than from the minds of the implementors.

The key idea needed to explain all the preceding examples, whether describing success or failure, is how effectively information is exchanged among all the people using or providing software components. Database technologies and international standards for the exchange of product data [1] have been widely identified as necessary in reducing the cost of software development and maintenance. We take it for granted that these technologies will be required to help users find and apply software components efficiently. We are left to wonder how the Linux operating system, which has been created entirely by a globally distributed, unstructured network of volunteers without the benefit of database technology or much in the way of software standards or formal quality controls, has become one of the most robust operating systems available. There must be lessons in the history of Linux and of the volunteers that created it for companies in the software component business, but precisely what those lessons are we will not venture to say.

3.2 COMPONENTS IN PROCESS MODELING

Among current process simulation tools we see a trend toward turning just one layer of the overall software system into components: process models and model solvers. The software system of the immediate future may be structured as shown in Figure 3-1.

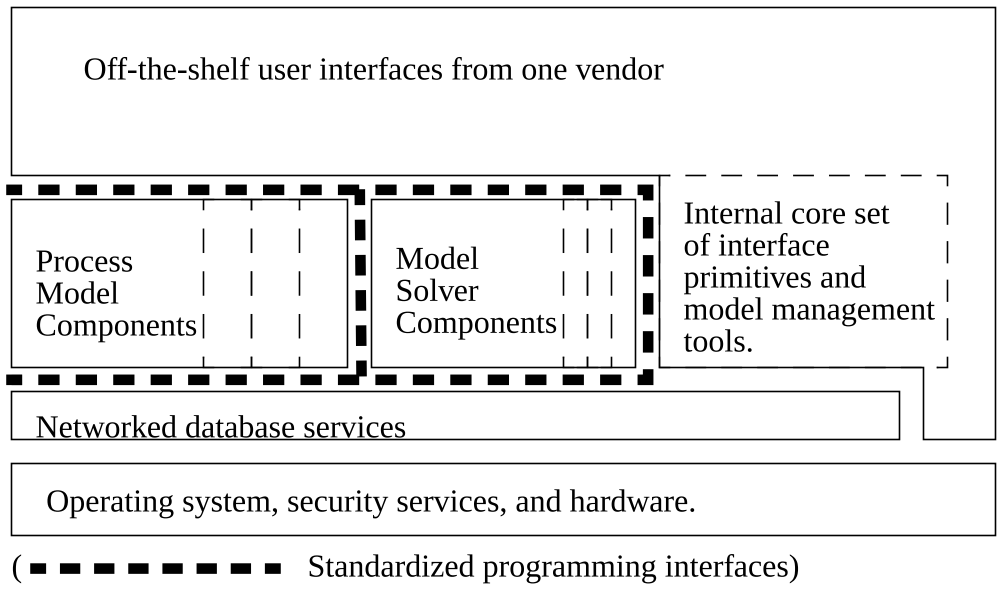
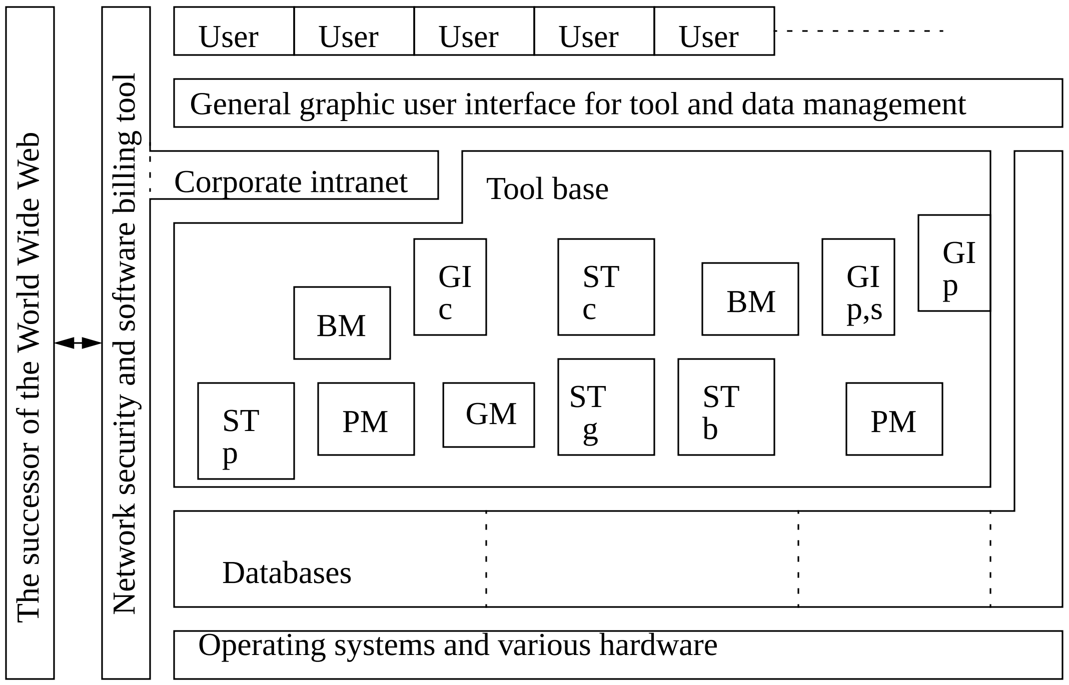

Off-the-shelf user interfaces from one vendor
Process Model Components
Model Solver Components
Internal core set of interface primitives and model management tools.
Networked database services
Operating system, security services, and hardware.
(■ ■ ■ ■ ■ ■ ■ ■ Standardized programming interfaces)

Figure 3-1 One engineering modeling system architecture

In Figure 3-1 the choice of off-the-shelf user interfaces probably fixes the choice of model management tools, of database services, and of operating systems and hardware. This architecture is expected to ease the long-standing problem of in-house engineering modeling efforts being tied to a single vendor's total system [9]. It adds very little by way of flexible support for the continuous change of today's dynamic business processes. Figure 3-1 bears a remarkable resemblance to the architecture of ASCEND III and, temporarily, to the architecture of ASCEND IV.

We propose that a more general modeling environment [20] is a more appropriate tool and that it should be structured more like the one shown in Figure 3-2.


User
User
User
User
User
General graphic user interface for tool and data management
The successor of the World Wide Web
Network security and software billing tool
Corporate intranet
Tool base
GI c
ST c
BM
GI p,s
GI p
BM
ST p
PM
GM
ST g
ST b
PM
Databases
Operating systems and various hardware

Tool key: BM = Business model, GM = Geometric model, PM = Process model, ST = Solver tool, GI = Graphic representation tool
Data key: b = business, c = combined, g = geometric, p = process, s = solver

Figure 3-2 A General Modeling Environment

Each block in the tool base is a computational model (the types BM, GM, PM are merely representative of possibilities, not an exhaustive classification), a data transformation tool (the type ST exemplifies one class of transformation), or a graphic representation tool (GI). Each tool and model is constructed according to internationally accepted software standards and might even be constructed from several other tools and data models. All tools and models can be queried in standardized ways to determine what they represent, what is required to use them properly, and so forth.

For example, the tool labeled “GI-p,s” might tell us that:

- it contains a graphic flowsheet creator capable of managing data from process models (p) and solving tools (s).
- we will automatically be paying license fees to company H any time a new flowsheet is saved to our database.
- the in-house history of calls to the toll-free hotline for company H is available in our database.

Since the tool knows about our billing practices and the content of our database, we surmise that this graphic interface tool has been constructed by someone in-house to manage a number of solver, diagnostic, and process modeling components we are not being billed to use and a commercial graphic flowsheet renderer licensed from company H. If we ask the database for more details about how this tool was added to the tool base, it will tell us.

Each tool base block is treated as potentially intelligent and capable of handling some classes of data autonomously, including data about the other tools with which it is designed to interact. For example, the general graphic user interface is not very intelligent about solving mathematical problems, but it is an expert in finding tools or models and in helping users assemble tools and models from existing parts in the tool base (our kit of modeling, solving, diagnostic, and representation components which covers many technical and business domains). In our hypothetical business, the billing and security tool will have veto power over tool-tool interactions, user-tool interactions, all interactions that go outside the company, and maybe even user-user interactions inside the company. What is important here is that the security and billing tool is provided by a vendor in that business, not one in the modeling business.

Each user has an environment which puts an organization relevant to her short and long term tasks on the models and tools of interest, ignoring all others. Our new employees are given a typically configured modeling software environment or the environment of the last person in the same position and a one page handout explaining how to search and navigate our tool base. Each tool in the tool base is able to at least minimally explain itself, and of course there may be tools created by other employees that explain typical tool-tool interactions they have discovered.

We would design modeling environments with the view that employees should feel in control of the modeling environment rather than vice versa. We would build environments (or rather our employees would) with off-the-shelf components for every routine aspect of the total business process including the routine activity of constructing non-routine models and tools. Consider the tools Herb uses in the following example:

> 7:45 a.m.: Herb arrives at work in his process plant near Kansas City.
>
> 9:30 Herb finishes disposing of his electronic mailbox. The task that has risen to the top is to help find out why reactor 100 at the Houston plant is still making off-specification material in spite of the best efforts of the local engineers and chemists. Herb's plant has a reactor built on essentially the same lines. Herb searches the company database for design and control models of either reactor and finds several. All of them treat the tubular reactor as an isothermal plug flow reactor with axial but not radial diffusion. The team in Houston has ruled out feed quality changes being the source of the problem.
>
> 10:30 After a general massaging of all the available data, Herb decides that a new reactor model is called for, using a full finite element (FEM) treatment to capture axial and radial diffusion effects and fine details of the reactor's internal geometry not previously modeled. He draws on mesh generation and other FEM solver tools he routinely uses [4] and a material properties model in use in the current reactor model. Using the FEM tools, Herb constructs a new reactor model, importing the required detailed geometric information from the most recently updated mechanical drawings in the Houston maintenance department. He specifies initial guess profiles by importing data from the best model the Houston team has been able to create.
>
> 10:50 The FEM solver says that the problem is not well specified, but does not have any bright ideas about how to fix the difficulty. The problem seems to be with some highly irregular geometry due to the newly modeled reactor internals and the specification of boundary conditions around those internals. Herb finds this odd since the FEM solver generally gives very good advice on boundary conditions.
>
> 11:00 After trying a few obvious permutations within the FEM input, Herb decides to try a tool from a simple flowsheeting system that knows how to do bookkeeping about equation and variable degrees of freedom (DOF). He opens up the flowsheeting tool, grabs the "Help on DOF" button on the flowsheeting system's graphic solver tool, and

drags the button into the FEM application. He puts away the flowsheeting system.

11:01 The general tool manager signals Herb that it was able to negotiate all the data exchange mechanisms required to make the FEM tool and the DOF tool work together.

11:05 The DOF tool immediately comes back to Herb with a message that temperature and pressure cannot be specified simultaneously in the material properties model at mesh nodes along certain boundaries. He makes a mental note to compliment the authors of the apparently simple flowsheeting system on their extremely good effort at making their tools reusable. Apparently, this particular material properties model was not anticipated by the FEM code developers. Herb modifies the boundary conditions, and sends a message to the FEM developers and material properties package developers about the poor interaction between their software packages.

11:30 Herb goes to lunch, having sent Houston his reactor model and pointers to the tools he used to make it. The Houston team will incorporate the model into a flowsheet simulator they have from company H and eventually solve the problem. A week later, the FEM company sends Herb a note thanking him for finding a subtle conformance error in their software, along with an updated copy of the responsible FEM-properties package interface component.

EXAMPLE 3-1 A successful component software enabled user

Frankly, the scenario in Example 3-1 sounds a bit futuristic to us as well, but this is what component software ought to bring. Creating such an environment is a big design challenge indeed, but one to which software designers and standards organizations are rising.

We in chemical engineering research should be rethinking our own tools to fit into such an environment, one where no discipline-specific tool is ultimately in charge. The user is in charge, with a host of software slaves (components) to assist, and the user understands that behind each slave is a person or company willing and able to improve it in order to keep the user's business. In the next section we will suggest some properties of open form models we think any user should insist on before buying them.

3.3 AN OPEN FORM MODEL COMPONENT DEFINITION

We believe all the following are included in the properties of an ideal open form modeling component. Many of these properties should be determinable from messages answered by an open form modeling software component, while some properties describe ways in which we would like components to interact with humans due to good design. We have considerable experience in open form modeling based on several reimplementations of ASCEND. We will start our definition with two examples of model reuse and a list of somewhat fuzzy design properties, then proceed to more specific attributes and messages.

3.3.1 EXAMPLES OF DESIRED COMPONENT REUSE

We seek a systematic way to allow component reuse and to avoid reinventing the wheel. Let us reconsider the chemist, the design engineer, the control engineer, and the operation planner introduced in Section 1.1, where each has separate tools. Let us imagine a scenario where they duplicate much less of each other's work, as illustrated in Figure 3-3.

The chemist creates a bench scale continuous stirred tank reactor model (Figure 3-3 C) containing a part which models the detailed kinetics and thermodynamics of some novel reactions. A design engineer works out a separation and recycle scheme (Figure 3-3 D) given only the list of chemical species involved. The design engineer models the reactor using a simple species mass-balance, because insufficient reaction information is available. Based on preliminary design reports and promising laboratory work, the project continues, eventually reaching the stage where detailed control models need to be constructed to investigate process safety and other issues.

A tubular reactor model is constructed using a collocation method, and at each point in the collocation mesh the chemical kinetics and thermodynamic part of the chemist's model is reused (Figure 3-3 B). The control engineer substitutes more rigorous dynamic models of the distillation columns and adds models of the furnaces which heat the tubular reactors. The results from the shortcut distillation design models are used to initialize the rigorous models, but the shortcut models are discarded so far as the control engineer is concerned.

Many months later the plant is actually brought on line, and optimal planning and scheduling models are needed. The plant model (Figure 3-3 A) is constructed reusing the control model *and* the design model. For optimization purposes, the planner adds models of the varying feed stocks and reuses the tubular reactor model and the shortcut distillation models, instructing the computer modeling system to disregard the conflicting simple reactor model and rigorous distillation models.

Each of the workers in this scenario may be using tool interfaces and modeling languages which are different. Because each modeling language and interface has been designed to manipulate and produce open, standardized mathematical models, each worker can reuse the relevant components of the models produced by previous workers. No doubt even the chemist reused models for thermodynamic calculations supplied by someone outside the illustration.

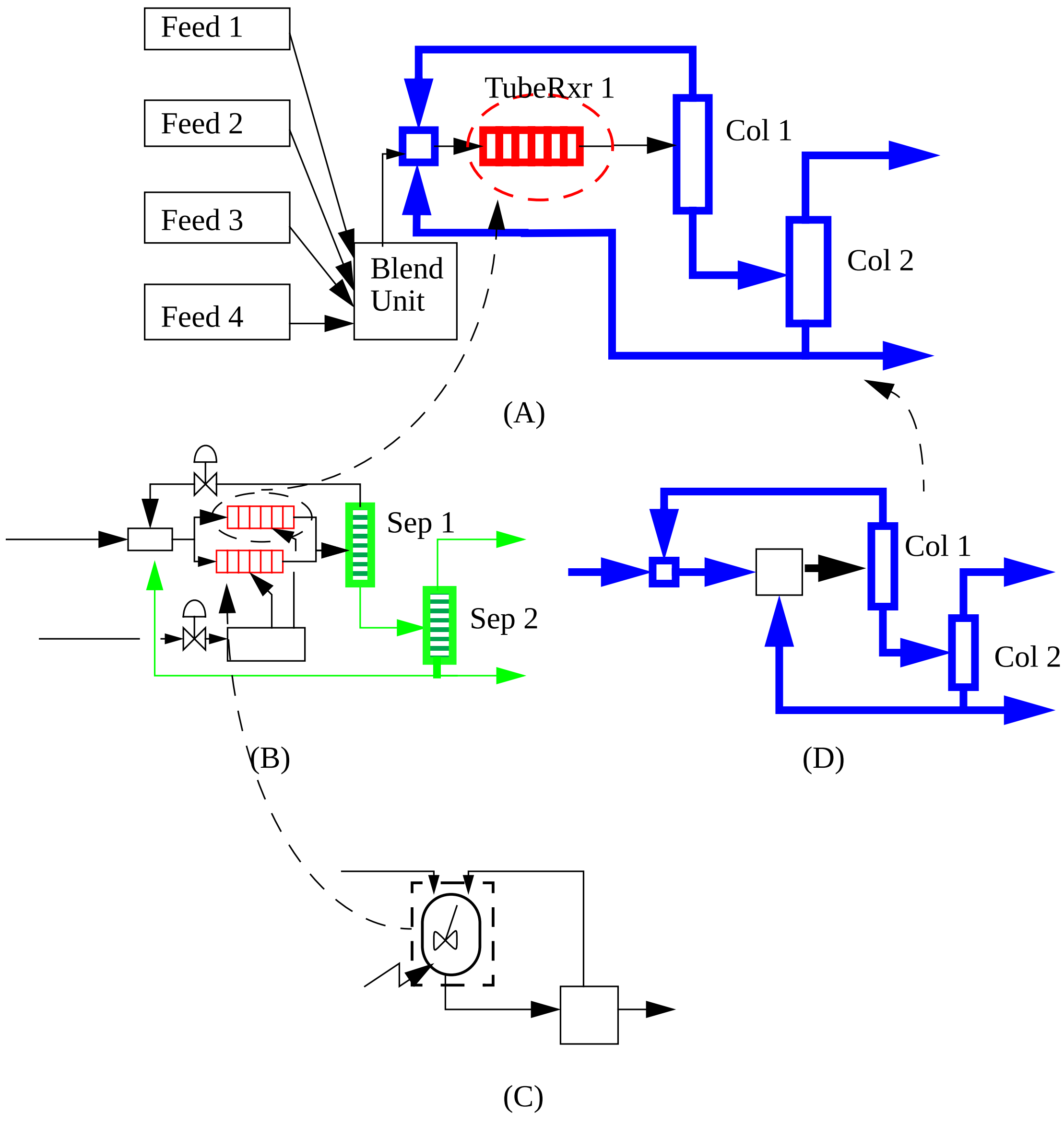


Figure 3-3 Production planning model (A) constructed from process design (D) and control (B) models. Reactors in (B) are defined including parts of the chemist’s bench reaction-separation model (C).

3.3.2 DESIGN PROPERTIES

We should be able to easily build a standards-compliant modeling component from other such components. In one case, a general component designed to calculate mass balances of a certain type (say, a distillation column) might specify that the user must supply complex subcomponents for physical property calculations and tray hydraulic calculations. The standards should include mechanisms that allow a component to verify user-supplied subcomponents at run time for semantic appropriateness. This kind of checking would ensure that we do not accidently send our distillation model a material properties model for molten steel. Such checking generally will involve checking logical constraints on the values of character string or integer classification codes, but might also include checking the expected values of real parameters against ranges of applicability as in the case of the aforementioned distillation model.

We should be able to test a new model easily for correct behavior. Independent validation of complex mathematical objects can be extremely difficult, since the likelihood of errors in the validation methods are often greater than in the object being tested. Validation is doubly difficult because constructing mathematical models frequently requires approximations which are expected to fail under all but a particular set of conditions. Modelers seldom manage to explicitly state the exact set of valid conditions on the first try. Therefore, we should expect each modeler to define each model with an initial set of self-validation routines which can be invoked by the end-user who defines a final application. We should be able to add additional testing routines easily as we encounter additional failure (or success) modes of the model.

Because we reuse models to build models, for example building a distillation column model out of tray models, provision should be made for a group of model instances that are bound together in another model to share as much of their implementation overhead as possible. In our distillation column, most of the equations on most of the trays can be evaluated using the same functions, since only the variable values and equation residuals are distinct among trays.

We should be able to distribute a model easily to different sites and different software systems once we create it. A model is most easily distributed if it is packaged in a small format and can be easily instantiated wherever it ends up. Basically, this means models must be distributed as standards-compliant source code since the computing community is not willing to agree on a universal binary format. Express [11] may end up being a language for such sources, but its universal utility and convenience is not yet well established. Even if some universally computable and applicable language is available, however, it will probably not help chemical engineers and other domain specialists express models in convenient ways because broadly applicable languages invariably have either enormous vocabularies with subtle shades of meaning or very small vocabularies with meanings too primitive to express modeling ideas efficiently.

We should be able to incorporate any application easily into any environment. That is we should be able to pull any part out of any model and reuse it in any other appropriate context. We should also be able to take any tool for manipulating models from any environment and incorporate it into any other, just as in Example 3-1 where Herb moved the “Help on DOF” button from his flowsheeting system to his FEM system. In adding this requirement, we begin to blur the line between models and modeling tools. This is as it should be and fits the jumbled tools and models in Figure 3-2. We will have more to say about this blurring when it comes to methods associated with models in the next section.

3.3.3 MODEL COMPONENT ATTRIBUTES AND MESSAGES

We need to choose a standard way of interacting with large composite models once they are assembled. We have found the following view quite serviceable. It is extensively expanded and evolved from ideas presented in [23]. It is more fully documented in [15,18,21, 16,22]. We present this view due to the shortage of descriptions in the open literature of the structures and functions used in process modeling systems which communicate with nonlinear solvers[1]. We believe it can serve as a basis for the discussion

1. The full details of our software are available in the source code which is distributed under the GNU License via http://www.cs.cmu.edu/~ascend/Home.html or can be acquired by sending e-mail to Arthur Westerberg, a.westerberg@cmu.edu.

and development of open standards for building large-scale, equation-based modeling tools.

Underlying this view is our reorganization and expansion of the solver application programming interface (API) of ASCEND III. One major goal of the reorganization is to demonstrate efficient reuse and extension of a large set of complex software tools by others[2]. Building on top of our ASCEND IV solver API, Kenneth Tyner has since been able to connect a sophisticated nonlinear solver[3], CONOPT [7], in less than a month. Vicente Rico-Ramirez was able to add extensions that support conditional modeling (the WHEN statement) to our basic solver API[4] very rapidly as well [18]. Tyner has also recently added new functions that allow the ASCEND IV graphic user interface code to remain ignorant of the details[5] required to support any particular solver while giving the user full control over these details via the graphical interface. Tyner and Rico-Ramirez' efforts have been and will continue to occur simultaneously and without significant conflicts; the success of their efforts suggests that our API design and implementation are fundamentally sound. Starting from our ASCEND foundation, they have not been substantially diverted from their primary research in order to build the tools needed to test the concepts and algorithms they are investigating.

We begin our description with an illustration of the information flows around model components, and then we proceed to more precise descriptions of the parts shown inside the SYSTEM object of Figure 3-4. Every step in the process is designed to allow (but not require) user intervention and inspection of intermediate results without substantial loss of computational efficiency. The implementation of these objects and data flows should be entirely in an object-oriented (data-hiding, message-passing) style, but our aim is to present a high level view rather than a dissection of the implementation[1]. Figure 3-4 shows how a number of distinct modeling tools (ASCEND and a foreign system, in this example) can produce information which is combined into a generalized mathematical

2. Primarily other graduate students.
3. A new solver client.
4. Rico-Ramirez modified our solver server.
5. A modern solver typically has dozens of control parameters, and some subset of these can drastically affect that solver's performance on different classes of problems.

representation (the system) suitable for reuse with a number of solving engines (CONOPT, LSODE, the ASCEND solver QRSlv) and user interfaces (ASCEND or GAMS).

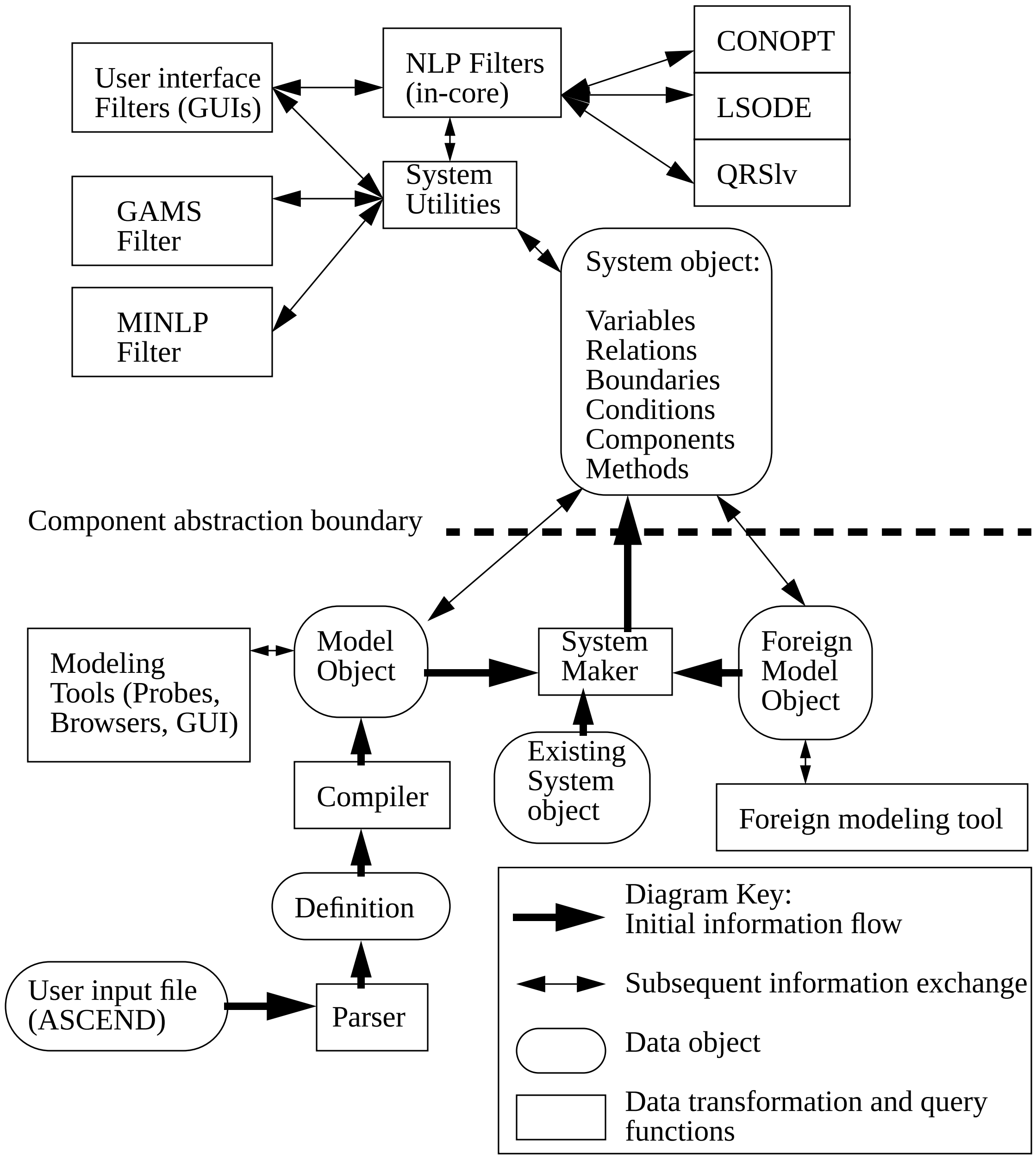


Figure 3-4 Reusable model and solution tools information flow.

Each filter transforms the information available via the system utilities into forms appropriate for clients communicating through the filter. The system object contains state information (values, mathematical relationships, component structures, and available methods) to support the mathematical component abstraction. All the clients of the system utilities and system object are insulated from the details of particular modeling objects, and each can potentially change the state of the system.

Each model object is defined by a tool appropriate to a particular task or particular user. The system maker performs the initial mapping from these diverse models (which might include another existing system!) onto a system object. We now turn to discussing the system parts, beginning with Table 3-1 which contains a synopsis of desirable system functions and data suggested by Drud and to which we add our own suggestions. While very few of the features in either our proposal or Drud's are completely new, this is, to our

Table 3-1: Desirable information in a reusable mathematical model

System feature	Drud proposal	Our additional suggestions
Structural relation-ships		component part/whole maps
Logical relations		relationships on sets and other discrete types
Boundary definitions		subregions of discrete and continuous variable spaces
Nonlinear relations	list of variables	scaling value
	residual	differential equations
	gradient	activity in conditional models
	hessian	identity of component context
	linearity	continuity
	bilinearity	monotonicity
	convexity	list of all similar relations
	concave/convex estimator	maximum additive term
	symbolic form	roots in one variable
Real variables	list of nonlinear relations	differential/algebraic status
	value	derivative of which variable
	fixed (.FX) flag	derivative with respect to which independent variable

Table 3-1: Desirable information in a reusable mathematical model

System feature	Drud proposal	Our additional suggestions
	scaling value	
	bounds	
	integer/boolean status	
	multiple names	
Discrete variables		symbolic, integer, boolean
		list of conditions controlled
		list of logical relations
Conditions		list of controlling discrete variables
		list of components controlled
		list of relations controlled
Subcomponent		list of available methods
		list of subcomponents (recursively)
Total system	Jacobian matrix	subcomponent list
	residual list	condition list
	variable list	
System utilities	simplifying equations	block lower triangularization
	remove singleton equations	degrees of freedom analysis
	tightening bounds	find variables near bounds
	expression elimination	find poorly scaled variables
	bounds on Jacobian elements	find causes of structural and numeric singularities
	primal and dual optimality condition information	part/whole information based matrix ordering
	interactive solvers[a]	matrix visualization
	derived constraints	code generation
	feasibility tools	

a. These have been a hallmark of ASCEND since its inception in 1985.

knowledge, the first time that anyone has proposed that they should be combined into a software system capable of being scaled up to handle interactively hundreds of thousands of equations.

We use one overall object (the system) containing lists of its variables, its constraints, its component-wise substructure information, and its conditional behaviors. For convenience, we call this object a solver system, but it is really just another software component. Each of the solver system’s variables, constraints, and so forth is an object that maintains certain state information and responds to messages. Each maintains state information because many different clients, including user interface tools, must interact with the solver system or any part of it during the initialization, solution, and analysis processes. All clients can, at least potentially, change the state of the solver system, and no client is responsible for managing the data structures associated with the solver system. Utility programs and solvers that interact with the system and its objects should not (and in ASCEND IV do not) need to know anything at all about how the solver objects are internally implemented. Not all of the functions described next are fully implemented in ASCEND IV yet, but the missing ones would be very easy to add.

Variables may be real or discrete, or even change between such roles as when solving a relaxed integer-linear program. We can ask each variable for a list of the constraints using it. We can ask each variable for a list of the conditional behaviors it helps determine. We can ask each variable if it is the derivative of another and, if so, of which one and with respect to which independent variable. We can ask each variable about its many other attributes, such as an appropriate scaling value, bounds, and whether it is currently specified or is to be computed. We can ask for the name or names of a variable in a system, but we cannot ask the variable alone because names require contexts to make sense. We will not give a catalog of possible variable attributes here, as the attributes we find interesting are only a subset of those that would be generally interesting. Curious readers should see [16] for our latest working definition of the variable abstract data type and operators.

Constraints may be continuous or discrete (logical rules), and they may be constraints that must be satisfied at the solution of the system, constraints describing boundaries within the system, or constraints that *might* need to be satisfied at the solution of the system depending on the conditional behavior of the system. Discrete variable values may be (and usually are) tied to the satisfaction of one or more of the real valued boundary constraints in the model, such as a temperature being beyond the critical temperature: $T > T_c$. We can ask a constraint for the list of variables it requires. We can ask a constraint for the list of conditional behaviors that control its activity. We can ask a constraint if it is the derivative of another constraint, and, if so, of which one and with respect to which independent variable. We can ask a real constraint for the value of its largest additive term [21] using the current or scaling values of its variables. We can ask a constraint for all the usual sorts of information such as residuals, gradients, lagrange multipliers, roots in one variable, linearity in subsets of variables, and convex underestimates [19]. Most importantly, we can ask a constraint for its symbolic form, as this is needed to help the less casual user debug open form models and to feed other symbolic tools. As with variables, constraints may have many other attributes, and the interested reader should examine [16] for our latest constraint abstract data type and operators.

The component-wise substructure information is useful for both debugging and decomposing (or reordering [2]) the equation and variable lists. Each component in the list may have its own component list; thus a system has an associated tree of subcomponents. Each component also defines a list of locally defined constraints. Any component at any level of the tree could be isolated with its own subcomponents for any desired manipulation, for example solving each tray in a distillation column before solving the whole column. Each constraint is associated with a single component, though variables may be shared among several components.

Conditional behaviors are simply logical structures that can be used to determine which of the constraints or subcomponents apply at the solution based on values of the discrete variables. We can ask conditional behaviors on which variables they depend, which constraints and subcomponents they control, and which subset of the controlled constraints and subcomponents are currently active in the problem.

Methods defined for one component may call on the methods of its subcomponents, since methods as well as variable data must all be visible to be reused effectively in an open form model. Each reusable component should provide us a number of methods that configure its variables (setting which are fixed, computed, and so forth) for various typical problems the component is expected to handle. Each component should provide a set of initial variable values that correspond to an easy to solve problem so that we can test it somewhat before plugging it into our larger model. Each component should provide solution checking methods that can be used to verify the correctness of both the assumptions and the solution when a solver pronounces the model solved. Each component should provide methods for rescaling variables. We should be allowed to add our own methods to a component so long as they do not conflict with its existing methods, and, when we are developing new components, we would like to be able to replace methods easily.

We need a method in each component for solving that component. Of course, the self-solution method should be able to use existing solver tools or to issue a message “Sorry, I can’t. No solver present.” response to our self-solution request. We can hardly make a distinction, then, between a component model and a solver in software terms. A solver is simply a component with some very interesting methods attached and no constraints or variables to contribute to the problem. Note that nowhere have we said the internal implementation of a method must be in all cases visible. This leaves a “solver” method free to be implemented in an arbitrary (but we hope efficient) way.

A model component should be able to identify where, if anywhere, we pay when we use it, and it should not perform any modeling for us until it is satisfied that we have indeed paid as necessary. Mechanisms for such financial transactions are being rapidly investigated [5,6], and we will not speculate further on how electronic commerce will eventually occur.

We need to be able to ask a component (and by recursion all its subcomponents) to produce pedigree information (when, by whom, and by using what tools it was created) so that we can track down the origin of unexpected and often quite annoying changes in behavior of a complex system when one of its components is replaced.

We need to be able to ask a component for examples of successful applications. It may respond with a message to look in some other specific place, or it will have some examples built into its on-line self-documentation. Similarly, we need access to failure and revision histories in a publicly (the paid-up user public, not the general public) accessible database uncensored by the component vendor. The component should include methods or instructions for accessing this database. The vendor retains a measure of control over the database by being able to add reviews of failure reports to the database. Most users would only consider the reviewed problems and their resolutions, thus avoiding the opinions of the few who are overly fond of irrelevantly complaining. The database must be uncensored so that, when users believe their legitimate problems are being ignored, they can use the database as evidence in discussions with their own management or with higher level representatives of the vendor. Finally, the failure and revision history database would no doubt include listings of what consultants to call when training and extra help is needed to use the component.

Shortcomings of the objects described thus far include their inability to represent temporal logic directly in process models or to represent realistic spatially distributed models. We have never deeply investigated modeling discrete event system algorithms with the ASCEND system. Nor have we deeply investigated finite element modeling and its associated algorithms for dealing with enormous numbers of variables on unstructured meshes. In discretizing time or space, the assumption is generally made that computing solutions at any given time or mesh point is an already solved problem. We make the opposite assumption in ASCEND; we assume mesh generation and discrete event simulation are already solved problems. All three assumptions are clearly incorrect in the general case, so we propose that the specialists in each domain should keep doing what they do best and that those few in each domain with an interest in interdisciplinary research should work as a group towards establishing general standards for communicating between their diverse model types.

In Chapter 4 we turn to an examination of several modeling languages and characterize what we have learned from ASCEND III/IIIc and other systems regarding the difficulty of modeling and the design of modeling language.

[1] "ISO 10303 standard (STEP)." International Standards Organization. Standard for the Exchange of Product Model Data.

2] Kirk A. Abbott, Benjamin A. Allan, and Arthur W. Westerberg. "Impact of preordering on solving the newton equations for process flowsheets." *AIChE Journal*, 1997. In revision.

[3] Kirk Andre Abbott. *Very Large Scale Modeling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1996.

[4] Inc. Beam Technologies. "Pdesolve." http://www.beamtech.com/products/pdesolve, April 1997. Product literature indicates that modeling with partial differential equations is now very easy. (!?).

[5] L. Jean Camp, Michael Harkavey, Bennett Yee, and J. D. Tygar. "Anonymous atomic transactions." Technical report, Department of Computer Science, Carnegie Mellon University, 1995.

[6] D. Chaum, A. Fiat, and M. Naor. "Untraceable electronic cash." In *Advances in Cryptology – Crypto '88 Proceedings,* pages 200–212. Springer-Verlag, 1990.

[7] Arne S. Drud. *CONOPT - A GRG code for Large Scale Nonlinear Nonlinear Optimization Reference Manual*. ARKI Consulting and Development A/S, Bagsvaerd, Denmark, 1992.

[8] Arne Stolbjerg Drud. "Interactions between nonlinear programming and modeling systems." Working paper, ARKI Consulting and Development A/S, March 1997.

[9] Peter D. Edwards and Gary Merkel. "Impact of an open architecture environment on the design of software components for process modeling." In James F. Davis, George Stephanopoulos, and Venkat Venkatasubramanian, editors, *Intelligent Systems in Process Engineering: Proceedings of the Conference held at Snowmass, CO July 1995,* volume 92 of *AIChE symposium series,* pages 307–310. CACHE, 1996.

[10] David Flanagan. *Java in A Nutshell*. O'Reilley and Associates, Inc., 1996.

[11] Thomas R. Kramer, Katherine C. Morris, and David A. Sauder. "A structural EXPRESS editor." Technical Report NISTIR 4903, National Institute of Science and Technology, Aug 1992.

[12] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services." Technical report, Computer Sciences Department, University of Wisconsin, 1995. Preliminary report of UNIX testing results.

[13] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, September 1995.

[14] C. C. Pantelides and H. I. Britt. "Multipurpose processs modeling environ-

ments." In L. T. Biegler and M. F. Doherty, editors, *Proc. Conf. on Foundations of Computer-Aided Process Design 94*, CACHE Publications, pages 128–141, 1995.

[15] Jennifer L. Perry and Benjamin A. Allan. "Design and use of dynamic modeling in ASCEND IV." Technical Report EDRC 06-224-96, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1996.

[16] ASCEND Project. "ASCEND IV modeling environment." http://www.cs.cmu.edu/~ascend/Home.html, 1997. ASCEND IV is released under the GNU Foundation License and Warranty.

[17] Y. Reich, S. Konda, S. Levy, I. Monarch, and E. Subrahmanian. "Varieties and issues of participation and design." *Design Studies*, 17(2):165–180, 1996.

[18] Vicente Rico-Ramirez, Benjamin Allan, and Arthur Westerberg. "Conditional modeling in ASCEND IV." Technical Report EDRC/ICES TR 0-0-0, Engineering Design Research Center, Carnegie Mellon University, 1997. In preparation.

[19] E. M. B. Smith and C. C. Pantelides. *Global Optimization of General Process Models*, chapter }. Kluwer, 1995.

[20] Eswaran Subrahmanian, Yoram Reich, Suresh L. Konda, Allen Dutoit, Douglas Cunningham, Robert Patrick, Mark Thomas, and Arthur W. Westerberg. "The n-dim approach to building design support systems." *Submitted to ASME Design Theory and Methodology*, 1997.

[21] Kenneth Tyner, Benjamin Allan, and Arthur Westerberg. "Scaling nonlinear equations." Technical Report EDRC/ICES TR 0-0-0, Engineering Design Research Center, Carnegie Mellon University, 1997. In preparation.

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

[23] Karl Westerberg. "Development of software for solving systems of nonlinear equations." Technical Report EDRC TR 05-36-89, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA USA, 1989.

CHAPTER 4 TOWARD BETTER MODELING LANGUAGES

4.1 AN OPEN FORM MODELING SYSTEM

Modeling complex systems, or even a single complex system, generally requires integrating the efforts of specialists from diverse disciplines and subdisciplines over a large period of time, frequently over large distances as well. For example, creating detailed models of a new chemical process requires chemists, several kinds of engineers, marketing analysts, financial analysts, and of course the mathematicians, scientists, and software specialists who create the various methods and tools in use by all the others. If we take a diverse group of such specialists, say about ten of them, and ask each separately what a model is or what a design is, we are quite likely to get as many different answers. To use such a group, divided by a common language and possibly spread over temporal and geographic locations, more effectively we need to create tools that help the group members translate each other's speech and share each other's methods and expertise [16]. We need open form modeling tools to handle very large problems, to support concurrently performing several tasks from one or many users based on a single model, and to improve communications among users with a wide disparity in abilities and technical dialects. We assume that expecting all the different users to learn a common language is unrealistically expensive and that a practical open form modeling language must explicitly support communications with other modeling tools in a peer-to-peer fashion. If we assemble a collection of languages and other tools - where each is constructed to be very good at something and where all are constructed with the goal of promoting communication and reuse in mind - we find ourselves with a modeling system. This modeling system, not just the mathematical models in it, is open form.

Many have hypothesized that the very basic view of the world as a place full of interacting objects with many kinds of interrelationships (links) among the objects is a view which can be mechanically computed and could facilitate mapping between diverse languages and domains of thought [1]. Before adding yet another text-based language to the mountain of object-oriented languages currently in existence, we need to answer several questions to determine if this is a wise course of action.

The questions are: What is the language to represent? For what audiences is it intended? What are the properties of good representations that we should consider? and, given our answers to the previous questions, can we create a significantly better language than those currently available, one that fits well into an open modeling environment? (Of course we will eventually answer the last question in the affirmative since the resulting language is the subject of our next chapter.)

We seek to represent arithmetic, logical, temporal, and configurational constraints describing physical objects and processes, and we seek generally to represent connections with other complex tools that share little or none of our semantics. We seek a representation usable by and promoting communications among a set of audiences including, but not limited to:

- casual users of simulation tools with comparatively little domain specific mathematical or configurational knowledge.
- engineering domain experts with very little time and very little software knowledge.
- modeling and software experts who create modeling tools.
- mechanical experts (other computer programs) that can take a model in our representation and solve it, reason about it, interpret it, or translate it into such diverse representations as may be needed to help us efficiently use the knowledge contained in it.

We can use the insights of other workers regarding what constitutes a good representation. Here we present a list of the properties of good representations from [17].

a. Good representations make the important aspects explicit.....
b. They are complete. We can say all that needs to be said.
c. They facilitate computation. We can store and retrieve information rapidly.
d. They are transparent. We can understand what has been said.
e. They are concise. We can say things efficiently.
f. They suppress detail. We can keep rarely used information out of sight, but we can still get to it when necessary.
g. They expose natural constraints, facilitating some class of computations.
h. They are computable by an existing procedure.

We will use this list as a reference point for discussions in the remainder of this chapter and in the description of the new language we are implementing, ASCEND IV, in the next chapter. We will highlight some interesting conflicts within this list.

4.2 PROBLEMS IN MATHEMATICAL MODELING TOOLS

In this section, we will briefly review features and difficulties of several mathematical modeling languages and, where the problems stem from the underlying language design, problems with the associated modeling environments. We focus primarily on commercial tools because they are usually documented and usually have had some care put into making them usable, thus we can feel safer in drawing inferences about the underlying design assumptions and their results when we must do so. In the next section, we will analyze in greater detail the performance of the ASCEND III language as measured against this same list of good representation properties.

Some languages define mechanisms to make extending the language easy. Symbolic processing systems in particular tend to be extensible. One example among many is Mathematica [18], an interpreted language incorporating hundreds of different symbolic and numeric operators from virtually all realms of mathematics. Once the notation of Mathematica is learned, simple sequences of complex calculations or new algorithms can be specified in a high level language. Mathematica is very good for conducting experiments that involve complex transformations on rather small (generally a few dozen items), arbitrarily structured data sets. The language of Mathematica is very nearly a complete representation of every significant data type and operator in modern mathematics because extensions are constantly being added. Unfortunately, the large number of operators makes it very difficult to look at an arbitrary piece of Mathematica code and tell what it will do. Thus property (d) is sacrificed in favor of properties (b) and (e). The Mathematica language also allows us to reuse the result of any previous computation in the current computation. This supports properties (c) and (f), but when this feature is combined with the symbolic structure of the language, it requires computational overheads that are prohibitive in large problems. Mathematica is an excellent tool, but not for solving highly structured, large scale, routine tasks that can be much more efficiently computed without the overhead required in Mathematica.

Nonsymbolic systems can also be extensible. The Matlab environment includes such a language. Matlab [11] organizes its operators into tool kits, groups of related operators, to alleviate some but not all of the difficulties that come with having a complete language. Matlab is an interpreted language specialized for modeling and using algorithms that manipulate matrices and vectors in complex or real arithmetic. Routinely called functions can be compiled for better performance. Matlab is very like an interactive FORTRAN 77 where the typical user need not be concerned with memory management issues. For matrix operations on small data sets (maybe as high as a few thousand variables) the Matlab language is fairly complete (b), rather concise (e), and of course highly computable (h). The language hides completely the important physical aspects of a model, forcing the user to keep track of the physics of the problem by some other method and thus scoring poorly on (a) and (c). In a Matlab representation, the machine can store and retrieve matrix information quite rapidly, but this does not automatically mean we the users can store and retrieve physically meaningful information efficiently. Matlab overloads several basic mathematical operations, such as addition and multiplication, to handle vectors and matrices, making the meaning of a statement quite ambiguous if we do not have a complete understanding of the context in which it appears. Due to this overloading, the Matlab representation is often not transparent (d). Matlab keeps some intermediate results and offers immediate feedback about each of the user’s actions, in a spirit similar to that of Mathematica. In spite of its apparent drawbacks, Matlab is extremely popular among both academic and professional users who must do non-routine numerical work on a routine basis.

Unstructured or at best array-structured equation-based systems (sometimes called matrix generators) that aim to promote optimization model reuse by separating problem data from problem statements have been around for nearly two decades. GAMS [6] is one such system that is widely used. GAMS is a batch interpreted algebraic modeling language. Equations and variables can be grouped into sparse arrays defined on sets and can have only very short names. GAMS array and set notation makes model definitions highly concise (e). The GAMS language can describe a wide range of algebraic problem classes involving discrete and continuous variables in forms easily fed to a number of sophisticated algebraic solvers, thus it has property (h). GAMS is notably incomplete (b) in that it does not provide any support for modeling with units of measure, in spite of being intended to express models of physical systems. This makes subtle input data errors nearly impossible to detect, allows the user to create physically inconsistent equations, and (as the purveyors of GAMS readily acknowledge [6]) leaves the user with the burden of finding a set of units which will lead to convergence of the model when it is solved. The lack of structure in the GAMS representation makes it difficult to create a complex model or to retrieve information about a complex model by examining the code (properties (c) and (d)) due to the difficulty of creating meaningful names in one global name space.

In more chemical engineering oriented mathematical modeling systems, i.e. modular flowsheet simulation environments, virtually all the commercial languages are deficient in three of the properties. All are exceedingly incomplete because the number of possible complex unit operations is quite large and creating a general model of a unit for just one or two customers is often hard to justify. In most cases, the contents of the library *define* the language. Property (b) is unsatisfied, and this is one of the reasons behind the drive for standards in the process simulation area; customers want to be able to create one model and use it with any commercial simulator. Unit operation model statements are generally not at all transparent, (d), in the simulation languages we have seen [14,5,2]. Generally, a graphic user interface with handcrafted, intelligent unit configuration tools is called on to cover up the simulation language difficulties. Finally, present simulators hide virtually all of the details (algorithms and internal variables) of a unit object so that we cannot get to rarely used information if we should need it, thus the languages do not support property (f) very well. The equation-based simulator Speedup [2] fairs somewhat better on all points, in particular supporting dynamic models well, whereas many do not. Speedup is incomplete if considered as a language which best supports unit operations primitives rather than equations.

Another symbolic language group bears mentioning: the physical phenomena oriented modeling languages that propose to derive a mathematical model from statements of the physical principles involved. One such language [15] allowed the user to write complex object descriptions in terms of conservation principles, transfer mechanisms, and various object interrelationships that a machine could then reason about to obtain a set of equations or derive higher level abstractions. As best we can tell, such systems are not widely in use. This is unfortunate because such a tool might be very useful in the area of creating new models and organizing unfamiliar concepts into object class and component part hierarchies . We probably need another tool for mapping the results of such a reasoning tool into a language more suited to routine computations on a large scale.

4.3 PROBLEMS IN ASCEND III

We now analyze the representation properties, reflecting on the ASCEND III [13] system, to motivate the design of ASCEND IV. We note here that all properties are subject to design trade-offs in creating a usable modeling system. It is our intent to learn from ASCEND III about the problems we are addressing, not to criticize the creators of ASCEND III. ASCEND III does an admirable job of making the case that appropriate tools can be created to aid the expert designer in modeling novel problems. We first give a summary of the major ASCEND III operators to help illuminate the discussion in the rest of this section. In these summaries we set in *italics* the arguments to the operators. The operator names are given in bold, as they are throughout our thesis. The minor operators used for writing equations and handling sets are described in Appendix C.

- **REFINES**: MODEL *new_type* REFINES *existing_type*; This operator creates a new type by adding statements to an existing type.

- **IS_A**: *name* IS_A *type*; This operator defines a part *name* in the context of the model and instructs the compiler to start construction of an associated object. The ASCEND compiler uses a multipass algorithm so the associated object may not be completed at the time this statement is seen.
- **IS_REFINED_TO**: *name* IS_REFINED_TO *more_refined_type*; This operator finds an existing object with the *name* given and changes the formal type of that object to the more refined type specified. The action of IS_REFINED_TO is called “deferred binding.”
- **ARE_THE_SAME**: *name1, name2* ARE_THE_SAME; This operator takes the objects associated with the names given and recursively merges those two objects into one. If one object is of a less refined formal type than the other, then it is first refined to the type of the other object. After the merge there is a single object known by both *name1* and *name2*.
- **ARE_ALIKE**: name1, name2 ARE_ALIKE; This operator takes the objects associated with the names and puts them in a list. Any time any object in the list becomes more refined by either IS_REFINED_TO or ARE_THE_SAME, every other object in the list is also refined. In other words, this operator automatically propagates changes in the formal type of one list member to all list members.

All of these operators, except the part declaration operator **IS_A**, are somewhat mysterious to programmers more familiar with non-object-oriented languages. We analyze these operators and other features of ASCEND on the basis of the properties of good representations, and then we look at some additional difficulties with reuse in ASCEND III.

4.3.1 THE PROPERTIES OF ASCEND III

“*Good representations make important aspects explicit.*” Three of the five operators for handling general objects in the ASCEND III language are ambiguous. The **ARE_ALIKE**, **ARE_THE_SAME**, and **IS_REFINED_TO** operators can all have unanticipated and difficult to track side-effects due to deferred binding. Common wisdom in the (small) community of ASCEND users is that one must instantiate a definition before one can tell what it really says. We use these three operators quite effectively to find a good first pass organization of information into inheritance and part/whole hierarchies. However, a first pass organization usually leaves far too many configurational constraints (rules on the way objects are wired together) implicit. For example, of about a dozen users of the reusable thermodynamics library created in ASCEND III, not a single one has been able to pick up the library and use it without a thorough explanation and example being delivered in person by one or another of the library's original designers. In at least four cases, even after the explanation, the users have chosen instead to write simpler one-off thermodynamic models because the reusable library was not easily reused or modified. Configurational constraints are quite plainly visible (or at least reliably checked) in most flowsheeting systems. The "simple" **ARE_ALIKE** constraint [13,8] has proved both insufficient and confusing to most users. Configurational constraints need to be made more explicit in any proposed language.

"*They are complete. We can say all that needs to be said.*" ASCEND III, as are most thesis-derived languages, is not intended to be complete. That the four users just described were able to create serviceable thermodynamic models in a very short time (all in less than a week) strongly suggests that ASCEND III is on the right track. We need many features missing in ASCEND III if we are to create an open form tool for general mathematical modeling use and reuse. We must somehow represent:

- boolean algebra.
- nonlinear algebra.
- equations for describing boundaries that partition a nonlinear space.
- equations placing constraints on object structures.
- selection among alternative object structures when compiling an object.[1]
- choice among alternative equation sets during nonlinear and discrete problem solving.
- temporal logic.
- equations defined by code external to ASCEND.[2]
- integer variables.
- character string variables.
- methods taking arbitrary actions on objects, including actions defined by external tools.[2]
- temporary variables (data structures) in methods.
- links to other languages, including human languages and graphic languages.

With this list of features added, we still have an incomplete language for the *overall* task of mathematical modeling. We are considering a language needed to be part of a bigger picture where other specialized languages are used for tasks around which they are

1. This item and the next were proposed as a single CASE statement in [13]. Apparently the confusion of these two very distinct kinds of conditional modeling stymied the ASCEND III developers.
2. Abbott [4] demonstrated this is practical in his thesis work on ASCEND IIIc.

designed; for example in deriving general formulae, Mathematica or other mainly symbolic systems are clearly better. We do not believe there is any need to create another symbolic manipulation language with hundreds of operators as a subset of a new language; our new language should be intentionally designed to be incomplete. What features to include is a practical matter, so we shall include only what we can expect to do well.

"*They facilitate computation. We can store and retrieve information rapidly.*" Both creating and executing complex searches of large structures of objects can be difficult. This is particularly the case with ASCEND III because each object can have several parents (be shared among several different contexts). Each ASCEND object stores a great deal of information about itself at considerable cost in memory (1200 to 2000 bytes per equation is typical when modeling a flowsheet) to speed the retrieval of commonly needed information. Unless object implementation is done very carefully, however, storing and managing extra information can end up costing more time than it saves. We must strive to design a language and implementation such that an interactive user working with any size model does not get frustrated with waiting on the system to compute any needed result.

"*They are transparent. We can understand what has been said.*" The ASCEND III language is not transparent. It is common knowledge among the ASCEND users at Carnegie Mellon that one must compile a definition, potentially to a very large or a very incorrect object, before one is able to tell what the definition really says. Similarly, the overall refinement hierarchy of an ASCEND library cannot be determined from the definition file without a great deal of effort. There are almost no elements of the language which look familiar to users from either the sequential-modular or the imperative programming paradigm, the majority of users. An object definition contains almost no clues about the authors intentions and how it ought to be used.

Comments are allowed in ASCEND, but, as with most computer languages, they are seldom an effective mechanism for documenting complex objects. Users routinely report being confused by library models that use **ARE_ALIKE** or **ARE_THE_SAME** unexpectedly. The omnipresence of deferred binding means the compiler cannot tell when an object has reached its final state, which makes determining error conditions (such as misspelling a variable name) and generating appropriate messages difficult to impossible; a name could be misspelled, or it could be the name of a variable to be defined in a later binding. Piela has noted [12] that users resort to tactics such as sending an incomplete object to a solver to find out what is wrong with the object. This suggests to us that perhaps the language needs to be rethought. The tool sets in the ASCEND III environment assume an object-oriented expert user with a good store of patience, a better store of math skills, and a great store of free time. Tools to support non-experts do not exist because most such tools require input information that is not computable from the language, such as the expected connections between objects. We need a new language that is highly transparent if we are to support a broad range of users and machine translation agents.

“*They are concise. We can say things efficiently.*” ASCEND III is sometimes too concise. **ARE_THE_SAME** and **ARE_ALIKE** are overloaded operators which allow very simple statements of very complex ideas that may require deferred binding side effects. **ARE_THE_SAME** was designed to allow connections between otherwise disconnected objects, for example merging a reactor output stream with a separator input stream. The primary use of **ARE_THE_SAME**, however, has been to create alternate names for variables down in nested scopes, for example creating a temperature, T, in a reactor model and then merging it with the temperature down in the vapor-liquid phase equilibrium model that is part of the reactor model. Here T is created only to be immediately destroyed in the merge. Such uses of **ARE_THE_SAME** account for up to 90% of the total time spent compiling a complex model, as we shall see in Section 5.3. When writing models in libraries that users expect to perform efficiently, we need ways to avoid compiling structures that will only be destroyed.

“*They suppress detail. We can keep rarely used information out of sight, but we can still get to it when necessary.*” We cannot tell what an object is for or how it got to be in its present form. This information is completely suppressed, though we generally need it to figure out how to use an object or why we cannot use an object as expected. We speculate that, if comments in ASCEND III were associated with keywords and could be produced in useful forms in the user interface, then the authors of models would be much more likely to produce well documented definitions because they would find the computable documentation useful themselves. We will enlarge on this idea in Section 5.4. There are many constant and set values in ASCEND III used to determine the detailed structure of a model (such as the number of trays in a distillation tower), and these values for the most part cease to be interesting once the model is created. We might like to have a general ability to suppress the display of these values as a feature of a new language, or we might decide to implement it as a user interface tool instead.

"*They expose natural constraints, facilitating some class of computations*." In the comments made on completeness, we have already listed the types of natural constraints that we find unfacilitated by the ASCEND III language. Perhaps we should note here that we want to model and solve large mixed nonlinear and discrete equation models which represent physical systems possibly discretized over time and spatial dimensions. We note, however, that some classes of equation models may require so much data that creating models in manually produced text files would be infeasible. We suggest that models with this characteristic are better handled in open form tools designed around the issues of data mountain manipulation, and that those tools and our language tool should contribute equations and variables as peer servers to an equation solving client.

"*They are computable by an existing procedure*." The ASCEND III language is not computable by an existing procedure. The procedures used to implement ASCEND III were coded in Domain Pascal on Apollo and HP workstations. The Pascal compiler needed to make ASCEND III computable no longer exists. This of course is not a surprising property of a code written in the late 1980's, but it suggests that we want a language that is implemented in a widely available, long lived, well-standardized language such as C. Languages which are still undergoing great change or are not properly standardized, such as Microsoft C++, should be avoided. ASCEND III was ported to C as ASCEND IIIc in 1993/4 [3], with its user interface implementation being completely discarded and redone in Tcl 7.2/Tk 3.4 [10]. The C portion of the port has withstood the hazards of 8 distinct UNIX systems[3] and countless compilers. The Tcl/Tk language is

3. IBM AIX, DEC OSF1, DEC Ultrix, HP UX, Linux, SGI Irix, Solaris 2.x, SunOS 4.

neither standardized nor stable, currently both are at version 8.0, and we have frequently wondered at the wisdom of this choice of interface languages. At present there is not a significantly better alternative available.

4.3.2 WHY REUSING ASCEND III IS HARD

As first mentioned in Section 1.3.1, we have observed that most modelers cannot reuse ASCEND models by other authors if merging and refinement of parts within the reused model is required for the new application. We can illustrate this with some simple pictures. In Figure 4-1 and Figure 4-2 each box is a model, and boxes shown inside boxes indicate that the inner model is a part of the outer model. Tags on the upper left corners of boxes indicate the kind of object the box represents. A stack of overlaid boxes indicates an array of similar objects.

In Figure 4-1 we see a structure typical of staged operations such as distillation.

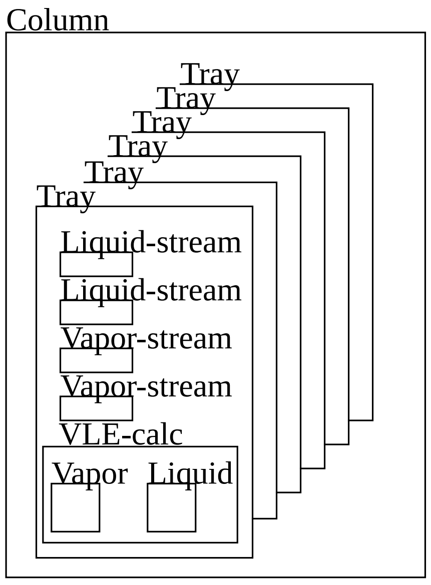


Figure 4-1 Part-whole structure of a distillation column.

If we want to reuse this distillation model, extending it to use a more detailed vapor phase part in the tray vapor-liquid equilibrium calculation, then we must find the Vapor part in the VLE-calc part in every Tray part in the Column and perform a refinement. To discover the changes needed, we must read through several nested definitions which contain many distracting details such as variable and equation declarations. Object-oriented users trained in information-hiding programming methods may have even more difficulty with this form of model reuse than those users versed in a FORTRAN style where everything is referenced through a COMMON block.

In Figure 4-2 we see structures typical of connected units in a flowsheet. The dashed lines

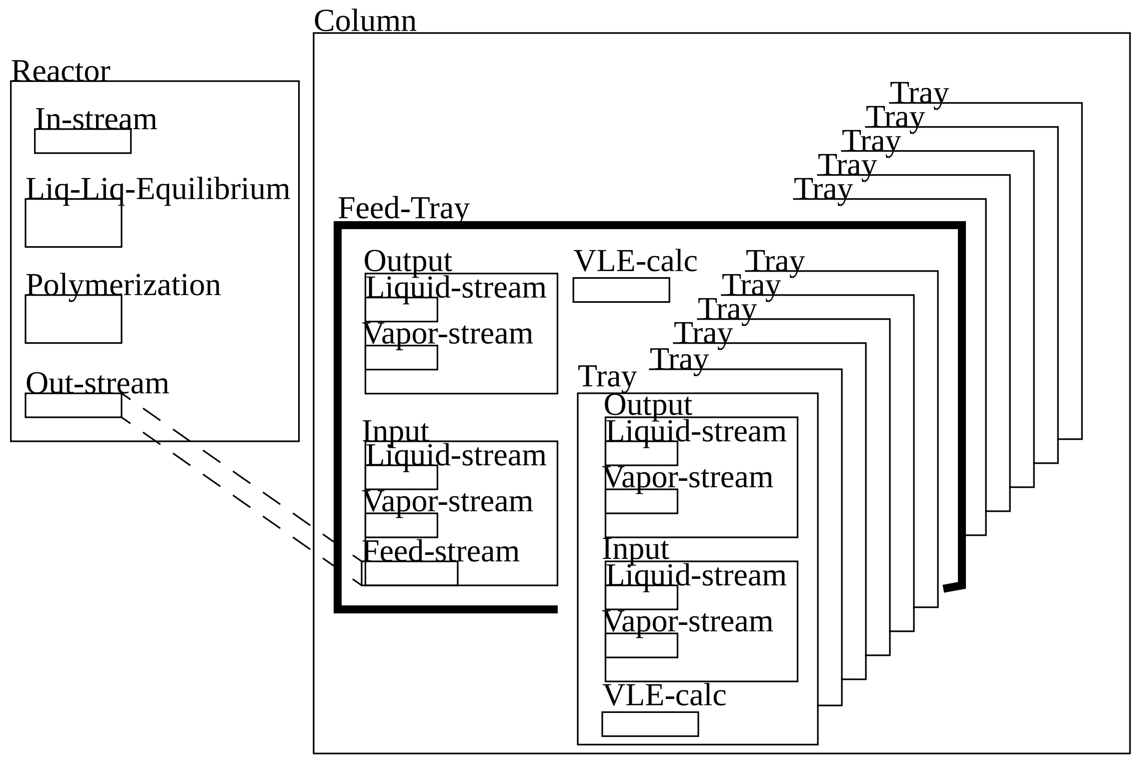


Figure 4-2 A connection between self-contained objects

indicate the part that must be shared between the two units. We must examine the code for the Reactor and Column very carefully before we can determine that the Reactor Output-stream must be merged with the Column Feed-Tray Input Feed-stream. We need to make the connection with a merge operation. Even if we successfully locate the corresponding stream part in each model, we still have no guarantee that the merge is possible.

An ASCEND III model does not automatically contain information which indicates the parts of the model that should be connected to other models in constructing larger models, nor does it automatically contain information about which parts of the model could be extended (by refinement) without invalidating the model. Experience has shown that adding comments to a model, no matter how thorough the comments are, is insufficient for most modelers to reuse the work of others. We need to find ways of modeling that make clear which parts of the models should be shared (merged or connected) and which can be extended or replaced without causing errors.

4.4 DESIRABLE LANGUAGE PROPERTIES

We now take what we've learned from existing languages and expand from our observations along various lines to expose additional features that should be considered if we are to create a language.

Other authors suggest that each operator in a language should have only one, clearly defined meaning [7], but there should not be so many operators that reasoning in the language requires constant reference to a dictionary. We wish to support nonroutine and preliminary modeling, however, so vague statements such as ASCEND III's **ARE_ALIKE** and **ARE_THE_SAME** may be necessary. We may need a few more operators in a declarative language than the five of ASCEND III. We have found that a small, unambiguous set of operators and an aptly named set of operands promotes effective analytic, synthetic, and algorithmic thought. We do not want to end up with the dozens or hundreds of operators in the languages like Mathematica and Matlab. The naming of operands (variables) is at the whim of the language user. We have seen that allowing descriptive names helps the user so a language should not place a small, arbitrary limit on the number of characters in a name.

For routine applications, we would like a language where there is exactly one way to state each meaning we want to express and that way should be obvious. Of course this idea conflicts with the notion of mathematical modeling where there are an infinite number of ways to write any equation or set of equations, and it conflicts with our desire to support modeling with incompletely understood information. We want to control the amount of information that needs to be simultaneously processed by human users of a computer language so they are not overwhelmed with extraneous detail. Information overload typically occurs in humans when there are more than about six distinctive items to be handled [9]. We believe an object-oriented language such as ASCEND III reduces the chances of information overload. We do not want to shield human users from excess information in a way that prevents machine users from accessing all the information in a model efficiently. We sometimes construct complex models with no good part/whole decomposition and no idea of which elements of the few dozen in the model will be most interesting to various end users, so ultimately we will need a tool which allows the user to hide types of information in specific contexts or globally until that user decides it is needed.

We would like to enable humans or machines to find out quickly what is common in and to abstract information from two definitions in a language. Similarly, we would like to spot differences easily. If we pick up a definition written in a computer language, we should like that definition to suggest uses and misuses for the object it defines. We would like the definition to be readable by a broad audience through barriers of discipline and time. We would like the definition to contain links describing how it relates to concepts in other languages. These links could contain information that does not compute in our language but might be computed by an appropriate client or server (human or machine) interacting with our language in a larger environment.

If we are to support open form modeling and environments, we need a language (or its resulting objects) which can be easily connected to other tools and their conceptual frameworks, either tightly or loosely depending on how well our concepts and theirs match. We need a language which provides clear, simple ways to adopt terms from other languages and use them in a manner consistent with both our language and the foreign language. In our language system we should be able to communicate as client or server with any tool which in any way deals with solving equations.

We should be able write a reusable definition which is self-checking: that is, if applied incorrectly, some explicit statement in the definition should be violated. Objects should be created from definitions and kept in a persistent database form because other tools or users may need to ask many arbitrary questions about a definition while learning how to interact with it.

We believe the ideas we discuss this chapter are sufficient to allow us to propose a new language (and some of the tools to support it). This language will significantly improve on those currently available. We believe that by extensively renovating, expanding, and pruning the ASCEND III language and modeling environment, we can create a reusable open form modeling language and point the way to the creation of an example of an open form tool suitable for the computer-aided engineering work environment of the future. We shall present the significant features of our new language in Chapter 5 and ideas for tools in Chapter 6.

[1] "ISO 10303 standard (STEP)." International Standards Organization. Standard for the Exchange of Product Model Data.

[2] *SpeedUp User Manual*. Cambridge, Massachusetts, 1991. Aspen Technology.

[3] Kirk A. Abbott, Benjamin A. Allan, and Arthur W. Westerberg. "Impact of preordering on solving the newton equations for process flowsheets." *AIChE Journal*, 1997. In revision.

[4] Kirk Andre Abbott. *Very Large Scale Modeling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1996.

[5] Aspen Technology, Inc. *Aspen Plus 8.5*, 1992.

[6] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user's guide*. Scientific Press, 1988.

[7] Elaine Kant and Allen Newell. "Problem solving techniques of the design of algorithms." Technical Report CMU-CS-82-145, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA USA, November 1982.

[8] R. Krishnan, P. Piela, and A. Westerberg. "Reusing mathematical models in ASCEND." In C. W. Holsapple and A. B. Whinston, editors, *Recent Developments in Decision Support Systems: Proceedings of the NATO ASI on Decision Support Systems*. Springer-Verlag, 1993.

[9] Allen Newell and Herb Simon. *Human Problem Solving*. Prentice-Hall, 1972.

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

[11] Eva Part-Enander, Anders Sjoberg, Bo Melin, and Pernilla Isaksson. *The MATLAB Handbook*. Addison-Wesley, 1996.

[12] Peter Piela, Barbara Katzenberg, and Roy McKelvey. "Integrating the user into research on engineering design systems." *Research in Engineering Design*, 3:211–221, 1992.

[13] Peter Colin Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[14] J. D. Seader, W. D. Seider, and A.C. Pauls. *Flowtran Simulation – An Introduction*. CACHE, 3rd edition, 1987.

[15] George Stephanopoulus, G. Henning, and H. Leone. "MODEL.LA. A Modeling Language for Process Engineering. Part I: The Formal Framework." *Comput. chem. Engng*, 14:813–819, 1990.

[16] Eswaran Subrahmanian, Yoram Reich, Suresh L. Konda, Allen Dutoit, Douglas Cunningham, Robert Patrick, Mark Thomas, and Arthur W. Westerberg. "The n-dim approach to building design support systems." *Submitted to ASME Design Theory and Methodology*, 1997.

[17] Lyle H. Ungar and Venkat Venkatasubramanian. *Artificial Intelligence in Process Systems Engineering*, volume III. CACHE, Austin, TX USA, 1990.

[18] Stephen Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley Publishing, 1988.

CHAPTER 5 ASCEND IV MODELING LANGUAGE

In this chapter we clarify our design goals, explain our decisions made in creating ASCEND IV, and present example results that convince us we are on the right path. We defer a detailed definition of the ASCEND IV syntax and semantics to Appendix C. We achieve the most critical of our results by defining an interface appropriate for construction of an open model. We can pass the objects which must be shared and the objects which the model reuser may refine through this interface. We can also pass constant parametric values through the interface. This form of model interface addresses the issues raised in Section 1.3 and some of the problems in Section 1.1. We explain this interface in Section 5.5. By itself, however, our definition of the model interface is insufficient to relieve the difficulties of reusing ASCEND III; we must define other user support mechanisms to achieve a total combination of interrelated features which is reusable. We define these supporting language features in Section 5.1 through Section 5.4.

We want a language which helps us create open form models quickly because modeling is simply an expense. We want a language which is intuitive and consistent; that is, we need one where the complete syntax is easy to figure out once we know a few examples. We want a language where operators are clearly defined to give us expressive power, user time efficiency, and assurance that we will get what we think we have said when writing a definition. We want a language that facilitates connections to other languages, computing tools, and users. We want a language in which we can say things unambiguously, yet we want to use it in extremely ambiguous situations such as developing new flowsheets or developing new models of other sorts. We want to satisfy most of the representation criteria listed in Section 4.1 simultaneously. We want to have our cake and eat it, too.

5.1 USER SPEED AND SYNTAX CONSISTENCY

Using ASCEND IV, we need to create or understand a large, complex mathematical model quickly where all the details may be examined as needed (by ourselves or by the computer) to support our decision making processes. Since the entire model is to be examinable simultaneously, we need an efficient way of naming every part. The array and dot notation of many computer languages [5,14] and also used in ASCEND III is sufficient. For example, the name of the temperature variable in the thermodynamic calculation in tray 100 of our column, `tray[100].thermo.T`, is quite clear. Similarly, all names of model types (classes) must be unique. For example, if we say `my_column` **IS_A** `high_pressure_column;` there must be only one definition of `high_pressure_column` in the modeling environment, or we cannot tell what `my_column` is. ASCEND IV checks for both part names and type names being uniquely defined when parsing a definition. It records type information about all names and issues warnings or errors about any names that are undefined or are used incorrectly. For example, ASCEND IV rejects at parse time definitions which attempt to merge two obviously incompatible objects. This makes it much faster for the user creating new models to detect and correct modeling errors which could only be detected during object compilation in ASCEND III.

Because ASCEND IV still supports deferred binding in some contexts via the **ARE_ALIKE** and **ARE_THE_SAME** operations, it is possible for a naming error to slip past the parse time checks. In particular, the existence and final instantiated type of specific array elements cannot be determined at parse time in reusable definitions which leave unspecified the value of the set over which the array will eventually be created. For example, in the general definition of a distillation column which leaves the number of trays unspecified, we cannot prove or disprove that the name tray[7] is valid. We have found that in practice such errors are rare; nonetheless, the compiler must still perform range checking when building objects.

We need to be able to compile or load routinely used ASCEND IV models rapidly and be able to modify their associated methods equally rapidly when the methods are insufficient for our purposes. In ASCEND IV we define the **ADD METHODS** and **REPLACE METHODS** commands which allow the user to experiment rapidly with interpreted method definitions. In ASCEND III, redefining methods requires destroying the objects created using the old methods and rereading the type and method definitions together. When experimenting with new large models, say a flowsheet, the ability to add or replace existing methods lets us save huge amounts of time that would otherwise be spent recompiling the objects. This functionality is only partially implemented.

We need to be able to tell whether order matters in a statement. When order matters is not clear from the syntax of ASCEND III or the extensions to it presented by Abbott [1]. ASCEND IV is a declarative language, and the order of statements made outside methods does not matter. Wherever the order of a list of names or statements in ASCEND IV matters, that list is enclosed in parentheses. For example, the order in which arguments are passed to most numeric functions matters. **SATISFIED**`(relation,tolerance)` returns TRUE if the residual value of `relation` is less than the absolute value of `tolerance`; clearly reversing the order of the arguments is incorrect. However, the order of arguments to ASCEND set operations **UNION**, **INTERSECTION**, **CARD**, and **CHOICE** does not matter. So, we write **UNION**`[set1,set2]`. In any place that an argument list whose order does not matter must be grouped together, ASCEND IV requires that square brackets be used. Two numeric functions, **SUM** and **PROD**, which specify the addition and multiplication of a list of numbers, respectively, do not require any list order since addition and multiplication are commutative. Hence we write **SUM**`[flow[inputs],flow[outputs]].`In ASCEND IV, the use of () and [] list delimiters is consistent.

The five general operators in ASCEND III are **IS_A, ARE_THE_SAME, REFINES, IS_REFINED_TO,** and **ARE_ALIKE**. As we indicated in Section 4.3, these correspond respectively to object construction, merging objects, type specialization, changing the type of an existing object (deferred binding), and grouping objects so that all are kept consistent with the same formal type. We expand this set in ASCEND IV to: **IS_A, WILL_BE, ALIASES, ARE_THE_SAME, WILL_BE_THE_SAME, WILL_NOT_BE_THE_SAME, REFINES, IS_REFINED_TO,** and **ARE_ALIKE.**

Most of the additional operators are used when defining parameters that will be passed to a model, as we explain in Section 5.5, so we follow a consistent pattern of changing the existing operators into their future tense verb forms. A parameter defined with **WILL_BE** will be constructed by the user of the model which needs the parameter. **WILL_BE** is similar to the forward declaration of most imperative languages [5]. We will discuss the semantic consistency of these parameter passing operators in Section 5.5.

The **ALIASES** operator defines a new name for an existing object, a concept which does not generalize any single ASCEND III operator, so of course it cannot be named consistently. We believe the name chosen makes the action of the **ALIASES** operator obvious.

We have named four new operators, but by following a consistent pattern of naming, we have avoided significantly expanding the conceptual space of operators an ASCEND IV user must manage in order to read or write a model. The addition of these operators and a parameter list for models makes it possible to create highly reusable, quickly compiled library definitions that minimize the application of deferred binding and avoid completely the ambiguous and expensive applications of ARE_ALIKE and ARE_THE_SAME operators. Several complete examples of such libraries are given in Appendix A. Detailed explanations of our other new operators are given in Section 5.5.

5.2 CLEARLY DEFINED OPERATORS

In this section we describe with examples several new features in ASCEND IV which we believe make a quicker, clearer understanding of statements and complete models possible. We also note some remaining difficulties in the language.

Using ASCEND IV we need to state efficiently and clearly all the constraints of a model, whether they are arithmetic (operating on real, integer, logical, or set data) relations to be solved or relations that define boundaries. Boundary equations are just like relations to be solved except that they are marked with a read-only boolean attribute, $boundary, with value TRUE. We use the **CONDITIONAL** statement to identify boundaries.

Code 5-1
```
MODEL pipe_flow;
      laminar IS_A boolean;
      Re IS_A reynolds_number; (* 0 < Re < INFINITY *)
      CONDITIONAL
            low_reynolds: Re <= 2100;
      END;
      laminar == SATISFIED(low_reynolds,0);
END pipe_flow;
```

We have expanded ASCEND IV to support logical relations [12], adding the logical equality (**==**) and inequality (**!=**) operators. Logical equations may be written over boolean variables using the boolean operators AND, OR, and NOT and boolean functions. One such boolean function is the **SATISFIED** function described in Section 5.1. We enforce a strict separation of equations into those in terms of real variables and those in terms of discrete variables. We do so because it makes models easier to read and understand, because it makes ASCEND IV much easier to implement, and because we have a new syntax (the **WHEN** statement, Code 5-5) to handle conditional modeling in a much more general way than mixing logical operations into real relations.

Sets are used to define the elements of an array in ASCEND IV. In ASCEND III, array elements can be defined arbitrarily meaning an array can change size at any time due to deferred binding. If we are to ensure model structure consistency and name uniqueness, we cannot allow arrays to change size once they are created. Also in ASCEND III, multidimensional arrays are always created with rectangular shapes, which does not reflect the sparsity which commonly occurs in the data of physical models. In ASCEND IV sparse arrays may be created by using appropriate **FOR** statements and set definitions.

Because the full expressiveness of ASCEND set notation has not been well exploited by most users and because the creation of sparse arrays is best explained by an example, we give an example of both in Code 5-2. The example is heavily commented so that it can be read and understood even if isolated from the surrounding text.

Code 5-2

```
MODEL rxntest;
(* this style is generic to all reaction systems that conserve mass *)
(* all the species in our system *)
      species IS_A set OF symbol_constant;
      species :== [‘A’,’B’,’C’,’D’];
(* all the chemical reactions we will model *)
      rxns IS_A set OF integer_constant;
      rxns :== [1..3];
(* the species participating in each reaction *)
      reactants[rxns] IS_A set OF symbol_constant;
      reactants[1] :== [‘A’,’B’,’C’];
      reactants[2] :== [‘A’,’C’];
      reactants[3] :== [‘A’,’B’,’D’];
(* reactions each species participates in *)
      reactions[species] IS_A set OF integer_constant;
      FOR i IN species CREATE
            reactions[i] :== [j IN rxns SUCH_THAT i IN reactants[j]];
      END;
(* sparse stoichiometric matrix. *)
      FOR j IN rxns CREATE
            FOR i IN reactants[j] CREATE
                  nu[i][j] IS_A integer_constant;
                  (* mole i/mole rxn j*)
            END;
      END;
(* production rates of all species *)
      production[species] IS_A molar_rate;
(* reaction rate for all reactions mole rxn j/time *)
      rate[rxns] IS_A molar_rate;
(* generic conservation of species equations *)
      FOR i IN species CREATE
      netgeneration[i]:
            production[i] =
            SUM[nu[i][j]*rate[j] SUCH_THAT j IN reactions[i]];
      END;
      x[species] IS_A mole_fraction;
(* rate coefficient values the user must specify *)
      k[rxns] IS_A real;
      FOR j IN rxns CREATE
      ratelaw[j]:
      rate[j] =
            k[j]*
            PROD[
                  PROD[ x[i] SUCH_THAT m IN [1..-(nu[i][j])]]
            | i IN reactants[j] ];
      END;
(* This equation filters out the rhs stoichiometric coefficients
* because [m..n] where (n < m) == empty set, [].*)
END rxntest;
```

Line 17 shows how to compute the row-wise sparsity pattern, `reactions`, from the column-wise sparsity pattern, `reactants`, of the stoichiometric matrix `nu`. We find the set operation syntax used to calculate `reactions` is one of the less intuitive (but much needed) features of ASCEND. We seldom find rigorous unordered set notation in a mathematical computing context which is odd considering that set theory underlies most mathematics. Line 22 defines the sparse array nu[species][rxns] so that the zero entries do not get compiled into objects. In ASCEND IV a sparse array can contain objects of any type. Line 44 demonstrates the **..** set operator. In ASCEND IV contexts where order matters, `m..n` will result in an iteration starting at `m` and *increasing* to `n`. In contexts where order does not matter, the **..** operator is simply a shorthand way of noting the integers ranging from `m` *up to* `n`, as noted in Line 48.

We cannot overstate the fraction of ASCEND users we have seen attempt to use **FOR** loops as if they were iterations in an imperative language. In keeping with the declarative nature of ASCEND, the statements in the body of a **FOR** statement are not executed in any particular sequence, nor is the index set that the **FOR** statement specifies processed in any particular order. We can demonstrate the problem with a very small example.

Code 5-3
```
FOR i IN [1..10] CREATE
      k[i] :== 2*i;                                   2
      wide_triangle[i][1..k[i]] is_a variable;        3
END;                                                  4
```

This **FOR** statement cannot be compiled because the constant assignment in Line 2 may not yet be done when we try to execute Line 3. We suggest, but have not implemented, replacing **FOR** with **OVER_ALL** in ASCEND IV so users will realize that ASCEND does not use the loops of traditional imperative languages to define model structures.

We express conditional compilation of declarative statements in ASCEND IV with the **SELECT** statement. We explain the particulars of the **SELECT** statement with an example, Code 5-4, abstracted from our thermodynamics library given in Section A.2.5.

Code 5-4

```
MODEL liquid_mixture(
      P WILL_BE pressure;                                         2
      T WILL_BE temperature;                                      3
      options WILL_BE liquid_options;                             4
      data[options.components] WILL_BE td_component_constants;    5
      correlation IS_A symbol_constant;                           6
) REFINES td_homogeneous_mixture(                                 7
      phase :== 'liquid';                                         8
);                                                                9
      pure[options.components] IS_A Rackett_liquid_component;    10
                                                                 11
      SELECT (correlation)                                       12
      CASE 'UNIFAC':                                             13
            FOR i IN options.components CREATE                  14
                  pure[i] IS_REFINED_TO                          15
                  RackettA_liquid_component(P, T, data[i],       16
                  options.pure_component_correlation['liquid'],  17
                  data[i].vp_correlation);                       18
            END;                                                 19
            UNIFAC_mixing_rule IS_A                              20
            UNIFAC_partials(P, T, V, H, G, scale, options,       21
                        phase, data, phi, y, pure, partial);     22
      CASE 'Wilson':                                             23
            FOR i IN options.components CREATE                  24
                  pure[i] IS_REFINED_TO                          25
                  RackettB_liquid_component(P, T, data[i],       26
                  options.pure_component_correlation['liquid'],  27
                  data[i].vp_correlation);                       28
            END;                                                 29
            Wilson_mixing_rule IS_A                              30
                Wilson_partials(P, T, V, H, G, scale, options,   31
                        phase, data, phi, y, pure, partial);     32
      END;                                                       33
                                                                 34
END liquid_mixture;                                              35
```

In Line 12 we put `correlation` in parentheses because the selection may be computed over an ordered list containing any combination of constant integers, booleans, symbols, or sets. In Line 6, we define `correlation` as a value supplied from outside the model, and it determines the structure inside the model. We compile all matching cases. The same name cannot be created in more than one case, therefore we name the mixing rule differently in Line 20 and Line 30. A name defined *outside* the **SELECT** statement, e.g. pure[i], can be refined differently in each case, as in Line 16 and Line 26. The names defined in unmatched cases might be accidently referred to elsewhere by the user or by a system tool, so we cannot just ignore the statements of alternative mixing rules. We must leave a place holder object for unselected parts of the definition so that the system can differentiate between objects missing because they were not selected and parts missing because the compiler has not yet been able to process their defining statements. We give the details of implementing the place holder in an efficient fashion in [12].

We express conditional inclusion of relations or model parts in a mathematical problem to be solved in ASCEND IV with the **WHEN** statement. We declare all the parts and equations we want to choose from dynamically and all the discrete decision *variables* which may be integers, booleans or symbols. We then write a **WHEN** statement, as shown in Code 5-5, which tells which equations and model parts to use when the discrete variables have particular values. We may include logical equations in the **WHEN** cases. Equations and parts specified in non-matching cases need not be satisfied at the solution of the overall problem. The details of implementing the **WHEN** statement are given in [12]. At present the ASCEND IV system uses these statements only when constructing a system (as described in Section 3.3) to determine which equations will be solved. We do not yet have a solver which simultaneously solves for the discrete and continuous variables.

Code 5-5
```
MODEL td_equilibrium_mixture (
      P WILL_BE pressure;                                              2
      T WILL_BE temperature;                                           3
      options WILL_BE multi_phase_options;                             4
      mix[options.phases] WILL_BE td_homogeneous_mixture;              5
      equilibrated WILL_BE boolean;                                    6
) REFINES td_alpha_mixture;                                            7
                                                                       8
(* Define the partial Gibbs free energy equilibrium condition. *)      9
      FOR i IN options.components CREATE                              10
            FOR j IN options.other_phases CREATE                      11
            equil_condition[i][j]:                                    12
                  mix[j].partial[i].G/data[i].G0 =                    13
                  mix[options.reference_phase].partial[i].G/
data[i].G0;                                                           14
            END;                                                      15
      END;                                                            16
(* The equilibrium conditions can be relaxed, decoupling the          17
* energy balance between phases. *)                                   18
      WHEN (equilibrated)                                             19
      CASE TRUE:                                                      20
            USE equil_condition;                                      21
      END;                                                            22
END td_equilibrium_mixture;                                           23
```

Using Code 5-5, we can set `equilibrated` to **FALSE** so the WHEN statement in Line 19 will prevent the equilibrium equations in Line 12 from being solved.

5.3 EFFICIENCY

We are aiming to improve the efficiency of the overall mathematical modeling process. This means, among other improvements, we need to make the compiler as fast as possible so that the user does not spend time waiting on the compiler during each iteration of the model construction process. In this section we explain two aliasing operators that we believe make possible a natural modeling style that has the added benefit of allowing more efficient compilation algorithms. We note some properties of refinement and universal objects that suggest we should minimize their use in modeling large or routine problems. We also propose language constructs new to ASCEND that allow the user or the system to represent concisely tabulated data, differential equations, and the attributes of variables.

In ASCEND III, nearly every name corresponds to a unique constructed object. An object can have more than one name in only two ways: if it is a **UNIVERSAL** object [3] or if it is the result of merging two objects. The **ALIASES** operator we described in Section 5.1 is equivalent to a pair of statements. We give an example of this in Code 5-6. In this example, our goal is to make local names `P1` and `P2` for the pressure in the reactor, `rxr`. The name `P1` comes from creating a pressure in Line 3 which we destroy by merging it with `rxr.thermostate.P` in Line 4. The name `P2` comes from waiting until the object `rxr.thermostate.P` exists and then just pointing at it in Line 5. The **ALIASES** operator has none of the deferred binding side-effects of **ARE_THE_SAME**, and it naturally indicates that `P2` is really just our local name for a component that belongs to the reactor. Not surprisingly, the compiler can execute **ALIASES** more efficiently than **ARE_THE_SAME** because it only needs to create one pressure instead of two. Of course, in this example we should define the name `P1` with an alias rather than merge it as shown.

Code 5-6

```
MODEL alias_example;
      rxr IS_A reactor;                                                  2
      P1 IS_A pressure;                                                  3
      P1, rxr.thermostate.P ARE_THE_SAME;                                4
      P2 ALIASES rxr.thermostate.P;                                      5
END alias_example;                                                       6
```

As models grow large and the objects being merged have more parts, the time we pay to merge objects grows at least quadratically. The time we pay to create an alias is small and is independent of object complexity, making **ALIASES** suitable for use in large models where efficiency matters.

Our example in Code 5-6 is trivial in order to be clear. In the ASCEND III ‘reusable modeling’ style which relies heavily on merging and refinement, the issue affects the construction of distillation and collocation models. We give brief statistics from a distillation example which highlights the problem, and then we look at result of our work for a collocation model. In a thirteen tray methanol-water distillation model we find complex objects with 273 names. An object with 273 names implies that 272 similar objects have been created and destroyed. Table 5-1 gives a sampling of these names.

By counting names we obtain the statistics in Table 5-2. These numbers suggest that the elegant, ‘reusable’ style of modeling in ASCEND III has a very high price in terms of lost work that is put into objects that are subsequently merged.

Table 5-1: Some names of water property data

Names (8/273)
tc.col.tray[7].input[‘feed’].state.mix[‘liquid’].pure[‘b’].data
tc.col.tray[8].VLE.mix[‘vapor’].data[‘b’]
tc.col.tray[13].vapout[‘vapor_product’].state.pure[‘b’].data
tc.feed.data[‘b’]
tc.feed.state.mix[‘liquid’].data[‘b’]
tc.col.tray[8].totfeed.state.data[‘b’]
tc.col.tray[7].input[‘feed’].state.mix[‘liquid’].data[‘b’]
tc.col.data[‘b’]

Table 5-2: Merge statistics for a 13 tray methanol/water column

Object Type	Number Created	Number Needed	Average Merges
Models	1,795	327	4.5
Arrays	12,754	854	13.9
Variables	149,573	1,902	77.6

We use arrays to group related items in ASCEND. Often we need an array whose elements comprise a group of already defined objects, such as in collocation formulae libraries [11]. In an ASCEND III collocation of a dynamic flowsheet model over *n* time steps, we must create *4n* flowsheet objects, only to destroy them while connecting the collocation steps. If we extend **ALIASES** so it can be used to construct arrays from similar objects, we can avoid this cost. Code 5-7 demonstrates constructing an array from already defined objects using the **ALIASES/IS_A** compound statement. This example is abstracted from Section A.2.10.

Code 5-7

```
MODEL euler_integration(
      nstep WILL_BE integer_constant;                                   2
      npoint WILL_BE integer_constant;                                  3
      grid[0..(nstep*npoint)] WILL_BE bvp_point;                        4
      n_eq IS_A integer_constant;                                       5
);                                                                      6
            (* model for solving boundary value problems *)             7
(* Create the little arrays for euler steps out of the user grid.*)     8
      FOR i IN [0..nstep-1] CREATE                                      9
            step_nodes[i+1][pset[i]] ALIASES                           10
                  ( grid[(i*npoint) ..( (i+1)*npoint)] )               11
                  WHERE pset[i] IS_A                                   12
                  set OF integer_constant                              13
                  WITH_VALUE ( 0 .. npoint );                          14
      END;                                                             15
                                                                       16
(* Create the total collocation equations using Euler intervals. *)    17
      FOR i IN [1..nstep] CREATE                                       18
            euler_step[i] IS_A euler(npoint,step_nodes[i],n_eq);       19
      END;                                                             20
END euler_integration;                                                 21
```

In Line 10 and Line 11 we construct a small array from user supplied `grid` elements. Each element contains a `bvp_point` model for calculating `n_eq` ordinary differential equations. In Line 12 we construct the set of subscripts, `pset[i]`, over which the array `step_nodes[i+1]` is defined, and in Line 14 we specify the elements of `pset[i]`. The ordered list of subscripts in Line 14 matches the ordered list of items in Line 11 one to one. There is no destroying of complex objects needed to assemble the integration model in Code 5-7. The `euler_integration` parameter list specifies that the user will provide an array of `bvp_points`, and the statements of the model distribute the points into the `euler_step` mesh models which define the collocation equations. We note that while the **ALIASES/IS_A** syntax is clear once written, users have pointed out that creating a correct **ALIASES/IS_A** statement is not a particularly intuitive process. We believe the functionality is correct, but admit that better ways of expressing it might be found with further investigation.

In Table 5-3 we see that, by avoiding merging through the use of the **ALIASES/IS_A** just described, we can reduce compilation times by an order of magnitude for general models containing large arrays of similar objects, such as collocation models and distillation models. The model tested in Table 5-3, `isomerization2` [16], is a collocation using the midpoint rule [6] with a total of 17 nodes (8 midpoint steps). Each node is a model of a two species, gas-phase isomerization reaction in the presence of an inert species. The details of this model are given in Section A.1.1. It contains 725 equations.

Table 5-3: Isomerization2 collocation model compilation times[a]

Case	ALIASES used	Copying used	Compile time (sec)
1	No	No	5.9
2	No	Yes	5.8
3	Yes	No	5.9
4	Yes	Yes	0.6

a. 110 MHz Sparcstation 5 running SunOS 4.1.4 and acc 2.0.1

What happened here? Neither the **ALIASES** we have defined nor the copying of similar model parts (in this case the grid nodes) suggested by Abbott [1] significantly reduced the compile time by itself. In case 1, an ASCEND III style compilation, the compiler destroys the 4n (n = 8) `bvp_points`, but many of them are destroyed before their complex substructures are fully assembled. Epperly notes this effect in [3]. In case 2, each `bvp_point` is fully assembled by copying a prototype, so 32 fully assembled `bvp_points` must be destroyed when the 32 merges are carried out. Copying the prototype saves us the time of reprocessing the `bvp_point` definition statements, but the time needed to merge fully assembled objects balances the savings. In case 3, we see that the use of **ALIASES/ IS_A** does not significantly reduce the compilation time, but, so long as it does not increase the time we are satisfied because our goal is to create a more reusable collocation model. We can almost tell what information we must supply to use the `euler_integration` model in Code 5-7 just by looking at its parameter list. In case 4, we see that the combination of copying prototypes and avoiding merge operations by using the **ALIASES** statement yields an order of magnitude speed up in compiling `isomerization2`.

In the cases of Table 5-3 where we used copying, we had to instruct the compiler ahead of time that copying `bvp_point` should be done. Large arrays of identical models are found in better understood models such as occur in libraries. We have observed (and routinely recommend to new users) that the arrays in poorly understood models ought to be kept small (under 10 elements) until the models become better understood. We suggest, therefore, that the compiler can use array size as one of the criteria for determining when to apply prototype copying methods automatically. By defining the ASCEND IV language as we have, we can avoid merging complex objects and capture information which the compiler can use to determine faster ways of building the objects we define. We can reduce compilation time by an order of magnitude, leaving the user with more time to spend applying the models to real problems. We have not yet implemented a compiler which automatically applies prototype copying methods to arrays of complex objects.

UNIVERSAL types in ASCEND never have more than one object associated with them. For example, `gas_constant` is a universal type which when compiled yields a `real_constant` with value 8.314 {J/mole/K}. All declarations using the `gas_constant` type share the same compiled `real_constant` instance. Each object keeps a unique list of all the contexts in which it appears, and maintenance of this list can become expensive in a large model where the universal object occurs thousands of times. Univeral objects minimize the memory required to store the universal information and ensure that the same information is used in all contexts since there are no duplicate objects. Some users create libraries containing universal typed variables for scaling equations[1]. Because each variable keeps a unique list of the relations in which it appears as well as the list of contexts in which it is defined, universal variables can cost large amounts of compile time and should be avoided where possible.

We want to enter array values efficiently as tabular data in ASCEND models. There is very little difference between what makes a table look good in ASCEND and what makes a table look good in any other computer language. Because tabular data entry is not a particularly new or difficult concept, and because it is not yet implemented in ASCEND IV, we refer the interested reader to Appendix C page 22 for our proposed definition of the

1. We note that this makes such libraries less reusable because one instance of a library model may need to be scaled differently from another, and the universal scaling value prevents independently scaling the equations of the two instances. This, however, is an issue of personal modeling styles.

ASCEND IV **DATA** statement.

Deferred binding (the **IS_REFINED_TO** statement and other statements that have refinement as a side-effect) is the primary source of objects with anonymous types in ASCEND. We discourage the use of deferred binding because of the potential inconsistency that anonymous types create and because deferred binding causes reconstruction of objects, a process which can be expensive. ARE_ALIKE is a particularly inefficient operator because it causes cliques of objects to be rebound any time one clique member is refined and because it frequently lulls new users into expecting the clique to be of equivalent anonymous type as well as equivalent formal type. This incorrect expectation can lead to very long periods of frustration spent debugging new models until the new user finally understands the inadequacy of formal type to ensure object consistency in an open system like ASCEND.

When we are developing very new models (such as a new thermodynamics library) where we do not know what are the right refinement and part/whole relationships, we can do our development and early testing on a small version of the model where the unscalabilities of **ARE_THE_SAME**, **IS_REFINED_TO**, and **ARE_ALIKE** are insignificant. If we decide to turn our new model into a scalable, reusable model, we can use the **ALIASES** and **SELECT** operators already described and the parameterized model operators coming up in Section 5.5 to rewrite the model in a more efficient manner. If producing efficient or general models is not high on our priority list, we can also give our small model and the examples of its application (both of which are in a very high level language) to a model efficiency or model generality expert for her to use as a detailed specification while creating an improved, reusable model. Because we can quickly supply a working prototype, we improve our chances of getting a model we really like back from the expert in a short time. If the expert returns us a model written in ASCEND, we can also compare how we model to how she models and learn a bit about efficient modeling, assuming, of course, that our supervisor gives us time to perform such a comparison.

5.4 EXTENSIBILITY

We have suggested in Chapter 4 that we should build our models and our modeling system from components that can be incorporated by other systems. Indeed, to allow complex model construction in reasonable time, we must [13]. We have suggested in Chapter 4 that we should build our models and our modeling system so we can easily incorporate tools and models from other systems. We want to construct such connections either tightly or loosely depending on how well the semantics of the connected components match. We do not want to end up creating a language with a thousand operators that no one could ever remember.

Decomposing the total ASCEND IV environment (200,000+ lines of code in three languages) into components usable by other systems is difficult due to the lack of published software protocols specifying the interfaces needed for object-oriented mathematical modeling components involving mixed logical and nonlinear equations [2]. We do not consider it further in this thesis because this component decomposition is also largely an ASCEND IV software implementation task not particularly affected by the modeling language syntax.

We can make decisions that enable or improve connections from our modeling language to external submodels, modeling tools, and users. Abbott [1] has described a language connecting tightly to externally defined submodels including external models that have component hierarchy information. Oh [9] has described a language connecting tightly to an externally defined model which discretizes a set of partial differential equations. Nilsson [8] has described a language connecting tightly to a graphical user interface tool. We now propose a more general formalism which we can use to forge loose connections in arbitrary languages, including human languages, by applying tools to model components. We view this ability as crucial to achieve the goal of future model component sharing.

We define **NOTES** in ASCEND IV. With **NOTES** we can associate a keyword/annotation pair with an ASCEND type definition, with names in a definition context, or with a library file of definitions. The annotating text is not understood by the ASCEND compiler, rather it is for the consumption of external tools and users. **NOTES** differ from comments in that the compiler stores them in a database that it can use to answer queries from outside agents. Each annotation is simply a block of text which can be of any length. ASCEND definitions, libraries, or compiled objects can be queried by external agents (users, graphic interfaces, other modeling systems) to retrieve these annotations. The annotating text might be the name of a graphic to represent a model, the contact information for user help, an explanation of the applications for the model, code in a foreign programming language appropriate to some task carried out with the model, or anything else needed to build bridges between the modeling process components, whether human or machine. We give an example in Code 5-8.

Code 5-8

```
MODEL liquid_feed_tray(
      in_stream[inputs] IS_A molar_stream;
) REFINES tray (
      inputs :== ['liquid','vapor','feed'];
);
      (* other statements omitted from example. *)
NOTES
      'bitmap'    SELF {feed_tray.bmp}
                  in_stream['feed'] {portdiamond.bmp}
      'basic'     in_stream['feed']
                  {The feed to this tray must be a saturated liquid.}
                  in_stream['vapor']
                  {The vapor feed to this tray should come from the
                  tray below in a vertical stack of trays.}
      'applicability' SELF
      {This feed tray model assumes steady state petroleum operation.
      For an unsteady state tray model, see dynamic_feed_tray in the
      HOLDUP library. For cryogenic work, see the library ICECOL.}
      'author'    SELF {Otto von Bismarck}
      'tech-support-usa' SELF {888-272-3634 (that's 888-ASCEND4)}
      'tech-support-www' SELF {http://www.ascend4.org/dist/help.htm}
      'revision' SELF {$Revision: 2.5.7 $}
      'competition' SELF {No one else is willing to sell only a tray.}
END NOTES;
END liquid_feed_tray;
```

We see in Code 5-8 that we can express much of the information we want in an ideal reusable model component as described in Section 3.3.3. The word **NOTES** introduces a block of annotations, and **END NOTES** closes the block. Each annotation in the block is stored in a database as a five element record {context, language-keyword, name, annotating-text, unique-id}, where context names a type or library, language-keyword is the name of the language in which the annotation is written, name is an ASCEND object name or **SELF** or **LIBRARY**, and the annotation is any text enclosed in {}[2]. If the name **SELF** is seen, we apply the annotation to the type definition or method in which the **NOTES** statement appears. If the name **LIBRARY** appears, we apply the annotation to the library in which the **NOTES** statement appears.

We can also annotate libraries and models from external sources. Code 5-9 shows examples of annotating a model and a library from a file loaded into the system later.

Code 5-9

```
ADD NOTES FOR TYPE liquid_feed_tray;
'in-house-expert' {Joe Glitsch, x-2742, jglitsch::oldvax.rnd.com}
'success'         {Works for the C10 fractionator in Red Stick, LA}
'failure'         {This model does a lousy job with light alcohols}
'failure-reviewed'      {The model works fine for light alcohols if you
                        use the thermo component from PineTree for VLE.
                        Reviewer: jglitsch}
'GUI-window-code' {win_liqfeed.tcl}
'demonstration-tcl'
{
      LOAD distill.lib;
      COMPILE testrun OF td_test_liqfeed;
      ASSIGN testrun.reflux 10;
      ASSIGN testrun.feed.totflow 1{mole/s};
      SOLVE testrun;
}
END NOTES liquid_feed_tray;

ADD NOTES FOR LIBRARY "cryostuff/ICECOL.lib";
'in-house-expert' {None. It's so easy to use that we can't be bothered
                  to understand the details.}
'success'   {Works for our air plant in Saskatoon.}
END NOTES ICECOL;
```

Sometimes the author of a model is not the best person to write explanations of it for general audiences, but we do not want people whose job is documentation to change the model accidently while they document it. By allowing the annotations to be made without touching the model source files, we can ensure that the process of documenting a model

2. If the annotation contains *unmatched* curly braces, "{" or "}", they must be escaped with the "\" character. Well-formed statements in most of the computer languages which use braces will be matched properly, so this is not a severe limitation. The other characters in an annotation are taken literally, including linefeeds and other whitespace characters; it is up to the external agent to interpret the annotation string.

does not change the model in any way. We can **ADD NOTES**, but we believe that policy should dictate that we cannot delete or replace **NOTES** once they are made.

We can make short notes at the time an object is defined using quotes. These short notes are all assumed to have the keyword `'is.'` For example:

Code 5-10
```
MODEL pump;
      inlet “the suction” ,
      outlet “the discharge”
      IS_A stream;
      rating “the design flow capacity”
      IS_A volumetric_rate;
END pump;
```

We see in Code 5-10 that these short comments are written after the name that they annotate. The implementation of this annotation facility is only partially complete, and we have not yet conducted extensive experiments on its use. We make proposals for its use in Chapter 6.

5.5 MODELING BY CONTRACT

We want a language which enables someone who has never before used an open form model library to pick it up and write an application model immediately without having to understand in detail the content of the library. We want a language which can perform logical consistency checks on structures and variables to ensure that the user of a library has not become a misuser of that library. We want a language that helps us rapidly understand the details of a library when we must extend it, whether we are an expert or not. And we want all this in a mathematical language which is general, not a language tied to one set of application concepts, with all the flexibility that ASCEND III demonstrates. In the previous sections, we have described many interesting features of ASCEND IV, but what is still missing is an easy path for a user from the more common flowsheeting or procedural programming language paradigms into this object-oriented, equation-based system.

Piela [11] and Abbott [1] both meditate on adding an interface to ASCEND models. Piela proposed creating several different interfaces for a given model where each interface is designed to help meet a specific user interest. By allowing ALIASES statements, we can easily write model wrappers (simpler models which hide the details of the more complex model being wrapped from the user) which have the same effect as defining multiple interfaces, and we can do so at very little additional computational cost. Abbott proposed creating a single parameter list which fixes all the structural parameters of the general definition so that a compiler could fully assemble the model once the parameters were specified, and he proposed creating new operators that build data structures by copying so that the user could force the compiler to use certain construction algorithms. Neither author investigated the propositions in any detail.

We think an appropriate question to ask when considering adding model interfaces to the ASCEND language is, “Considering that ASCEND IV is to be an experiment in helping people more efficiently use open form models and modeling software components, what can we do with an interface?” Most imperative languages [5,7, 4,10,15,14] use interfaces to *hide* information, which is the opposite of our goal: to make as much information as we can available to various human and machine agents with different needs without overwhelming any of them with it. We agree with Abbott that we should be able to know everything required to compile a model by looking at only the interface of the model.

We can do more, however, with the model interface. We can write statements to ensure the consistency of the arguments to the model. We can use passed objects and passed values to make the routine compilation of a model efficient. We can use model interfaces to help control the deferred binding events that generate incompatible anonymous types in a compiled object. We have defined **ALIASES** which can look down in the current scope. Model interfaces can give us a way of “looking up,” a way to state what parts any larger application context which uses our definition must supply before the compiler should bother building our model. Before we continue, we must more clearly define the operators named in Section 5.1 which we use in model interfaces. Basic model interfaces have the form:

> **MODEL** *model_name* (*passed_object_and_value_definition_statements*)
> **WHERE** (*preconditions_of_model_use_statements*);

passed_object_and_value_definition_statements is an ordered list of statements using **IS_A** and **WILL_BE.**

- **IS_A**: establishes a new name and a new object of any constant type[3] in the model. In short, **IS_A** defines a constant parametric value. The *value* of the object will be assigned by an expression supplied to the model from the application context. Later refinements of the model we are defining can also assign the value of **IS_A** defined parameters, removing them from the list of information the user must supply. For example, in Code 5-8, Line 4 `liquid_feed_tray` assigns the set `inputs` the value `['liquid','vapor','feed']` so the interface statement "`inputs IS_A set OF symbol_constant`;" which must have been in the less refined definition `tray`, is no longer needed.
- **WILL_BE**: establishes a new name in the model for an object compatible with the designated type that the application context will create and pass. This defines a passed object.

preconditions_of_model_use_statements is an ordered list of statements using **WILL_BE_THE_SAME**, **WILL_NOT_BE_THE_SAME**, and relations among constants. These statements correspond to the *assertions* [7] of “programming by contract.”

- **WILL_BE_THE_SAME**: asserts that named parts of two or more objects supplied to the current scope must in fact be *one* object.
- **WILL_NOT_BE_THE_SAME**: asserts that named parts of two or more objects supplied to the current scope must be *distinct* objects.

Logical and real relations, possibly written inside **FOR** statements, can be included in the model interface as preconditions of use. All the values in these relations must be constants and the relations must be satisfied before the model will be constructed.

We give a very simple example of our model interface in Code 5-11, a distillation column with one liquid feed and two liquid products. The model interface for `demo_column` follows the definition of `hexane_column_flowsheet`. For this example, we are concerned with simulating only a single distillation column, a task which is not infrequent in our industrial experience.

3. Parametric values have one of the types: set, symbol_constant, integer_constant, real_constant, boolean_constant. Arrays of these are also considered parametric values. In the parlance of ASCEND III, parametric values are “structural constants.”

Code 5-11

```
MODEL hexane_column_flowsheet;
      hexsep IS_A                                                    2
      demo_column(                                                   3
                  [‘n_pentane’,’n_hexane’,’n_heptane’],              4
                  ’n_heptane’,                                       5
                  13,                                                6
                  7                                                  7
                  );                                                 8
END hexane_column_flowsheet;                                         9
```

Lines 4-8 of this model are the parametric values passed through the model interface. The `demo_column` model is actually a wrapper for a more complex, more configurable model, `simple_column`. Both these models are given in Appendix A.

```
MODEL demo_column(
        components IS_A set OF symbol_constant;
        reference IS_A symbol_constant;
        n_trays IS_A integer_constant;
        feed_location IS_A integer_constant;
) WHERE (
        reference IN components == TRUE;
        n_trays > 5;
        feed_location > 2;
        feed_location < n_trays - 2;
);
```

To illustrate most of our model interface concepts in more detail, we present in Code 5-12 a very simple flowsheet containing nothing but a flash unit. We present a scenario describing how `test_vapor_liquid_flash` can be assembled from a flash library model by looking only at the parameter lists of the required models and responding to the diagnostic messages from the ASCEND IV compiler. In fact, in our scenario almost no domain specialist knowledge is needed; we can almost write the model mechanically by following the syntax of the language. Thus we demonstrate that the model interface can help bridge the gap between the experts who write libraries and the people who use them.

Let us assume a user who is new to the ASCEND language but who has some experience writing FORTRAN or C. He has a pretty good idea of what objects are for, but still tends to think of them as “subroutines” or “functions.” For the moment we will assume as insignificant the thorny problems encountered by new users in navigating the ASCEND

IV interface. In particular we assume that the libraries needed are all loaded and his task is to create a model. We have assured our user that his existing skills, the basic definitions of the operators **IS_A**, **WILL_BE**, **WILL_BE_THE_SAME**, **WILL_NOT_BE_THE_SAME**, and a willingness to make up reasonable names will be enough to accomplish his task, if he trusts the ASCEND system to help him figure out how to define these names.

Let us call our user Ed. With a point-and-click modeling interface much of what Ed will do could be simplified and automated. Ed will write a text input file, however, as the issues we wish to illustrate are clearly demonstrated in so doing. Ed wants to simulate a flash unit with one liquid and one vapor product. He looks at the list of loaded ASCEND libraries: system, atoms, components, thermodynamics, stream, and flash. Flash looks promising. He clicks on flash and sees `mass_balance_flash`, `vapor_liquid_flash`, `multifeed_flash`, and `VLL_flash`. He clicks on `vapor_liquid_flash` and sees the parameter list:

```
MODEL vapor_liquid_flash(
      feed WILL_BE liquid_stream;
      vapout WILL_BE vapor_stream;
      liqout WILL_BE liquid_stream;
      flash_state WILL_BE td_VLE_mixture;
) WHERE (
      state.heavy, liqout.state WILL_BE_THE_SAME;
      state.light, vapout.state WILL_BE_THE_SAME;
      vapout.state.T, feed.state.T WILL_NOT_BE_THE_SAME;
      vapout.state.P, feed.state.P WILL_NOT_BE_THE_SAME;
      liqout.state.options.ds.components ==
      vapout.state.options.ds.components;
);
```

So in his text file he puts:

```
MODEL test_vapor_liquid_flash;
      flash IS_A vapor_liquid_flash( feed, vapout, liqout, state);
END test_vapor_liquid_flash;
```

Ed is not sure what exactly all the **WILL_BE_THE_SAME**`'s` are about yet, but the requirement that the same components are in both product streams seems reasonable. Since nothing has suggested he must define the components yet, it seems safe to ignore doing so at this time. He sees from the model interface of `vapor_liquid_flash` that he needs to declare his process streams and a `td_VLE_mixture`. So he defines:

```
      feed IS_A liquid_stream;
      vapout IS_A vapor_stream;
      liqout IS_A liquid_stream;
      flash_state IS_A td_VLE_mixture;
```

Ed tries to compile his model, and it fails. The compiler complains that each stream model requires a passed object that will be locally named `state` and that `td_VLE_mixture` needs five passed objects. Ed browses the model interfaces of `vapor_stream`, `liquid_stream`, and `td_VLE_mixture` where he sees:

```
MODEL liquid_stream(
      state WILL_BE liquid_mixture;
) REFINES td_stream;
MODEL vapor_stream (
      state WILL_BE vapor_mixture;
) REFINES td_stream;
MODEL td_VLE_mixture(
      P WILL_BE pressure;
      T WILL_BE temperature;
      light WILL_BE vapor_mixture;
      heavy WILL_BE liquid_mixture;
      equilibrated WILL_BE boolean;
) WHERE (
      heavy.P, light.P WILL_BE_THE_SAME;
      heavy.T, light.T WILL_BE_THE_SAME;
      light.options.phase != heavy.options.phase;
) REFINES td_equilibrium_mixture();
```

Ed sees that, since all the streams and the `td_VLE_mixture` need mixtures, he should give them distinctive names, and he writes the following.

```
      equilibrated IS_A boolean;
      feed_state IS_A liquid_mixture;
      liquid_state IS_A liquid_mixture;
      vapor_state IS_A vapor_mixture;
```

As there seem to be model interfaces everywhere, Ed decides to check on liquid and vapor mixtures. Sure enough they also need to be passed some objects:

```
MODEL liquid_mixture(
      P WILL_BE pressure;
      T WILL_BE temperature;
      options WILL_BE liquid_phase_options;
) REFINES td_homogeneous_mixture;
```

```
MODEL vapor_mixture(
      P WILL_BE pressure;
      T WILL_BE temperature;
      options WILL_BE vapor_phase_options;
) REFINES td_homogeneous_mixture;
```

Ed changes what he just added to read:

```
      feed_state IS_A liquid_mixture(P, T, liquid_options);
      liquid_state IS_A liquid_mixture(P, T, liquid_options);
      vapor_state IS_A vapor_mixture(P, T, vapor_options);
```

and looking ahead adds:

```
      P IS_A pressure; T IS_A temperature;
      vapor_options IS_A vapor_phase_options(ds,’Pitzer’,’Pitzer’);
      liquid_options IS_A liquid_phase_options(ds,’Rackett’,’UNIFAC’);
```

because `vapor_phase_options` and `liquid_phase_options` show:

```
MODEL vapor_phase_options (
      ds WILL_BE td_component_data_set;
      component_thermo_correlation IS_A symbol_constant;
      mixture_thermo_correlation IS_A symbol_constant;
) REFINES single_phase_options(
        phase :== 'vapor';
        component_thermo_correlation == 'Pitzer';
        mixture_thermo_correlation == 'Pitzer';
);
MODEL liquid_phase_options (
      ds WILL_BE td_component_data_set;
      component_thermo_correlation IS_A symbol_constant;
      mixture_thermo_correlation IS_A symbol_constant;
) REFINES single_phase_options(
      phase :== 'liquid';
      component_thermo_correlation == 'Rackett';
      (mixture_thermo_correlation == 'UNIFAC') OR
      (mixture_thermo_correlation == 'Wilson');
);
```

which seem to suggest that he must choose from UNIFAC or Wilson in the liquid and is otherwise rather limited in his choice of correlations. At last Ed is down to just one new parameter to define, `ds`, so he types:

```
      ds IS_A td_component_data_set
```

and being used to looking just one step ahead, sees

```
MODEL td_component_data_set(
        components IS_A set OF symbol_constant;
        reference IS_A symbol_constant;
) WHERE (
        CARD[reference IN components] == 1;
        FOR i IN components CREATE
                (i ==  'hydrogen' ) OR
                (i ==  'carbon_dioxide' ) OR
                (i ==  'water' ) OR
                (i ==  'chloroform' ) OR
                (i ==  'methane' ) OR
                (i ==  'methanol' ) OR
      END;
);
```

He finishes the line with `([’methanol’,’water’],’water’);`. Ed tries to compile his model. The compiler catches a mistake. The compiler reports:

```
WILL_NOT_BE_THE_SAME statement contains identical/merged instances.
      vapout.state.T, feed.state.T WILL_NOT_BE_THE_SAME;
WILL_NOT_BE_THE_SAME statement contains identical/merged instances.
      vapout.state.P, feed.state.P WILL_NOT_BE_THE_SAME;
Parameter passing error: Merged instances found in WILL_NOT_BE_THE_SAME.
Error in executing statement:
      flash IS_A vapor_liquid_flash( feed, vapout, liqout, state); Line
2: vlftest.asc.
```

Ed lost track of this condition on the parameters of the flash, but it seems obvious now. He adds variables feed_P and feed_T to pass into the feed_state. Ed’s total model now looks like:

Code 5-12

```
MODEL test_vapor_liquid_flash;
      flash IS_A
      vapor_liquid_flash( feed, vapout, liqout, flash_state);

      feed IS_A molar_stream(feed_state);
      vapout IS_A vapor_stream(vapor_state);
      liqout IS_A liquid_stream(liquid_state);
      flash_state IS_A
      td_VLE_mixture(vapor_state, liquid_state, equilibrated);

      feed_state IS_A liquid_mixture(feed_P, feed_T, liquid_options);
      liquid_state IS_A liquid_mixture(P, T, liquid_options);
      vapor_state IS_A vapor_mixture(P, T, vapor_options);
      equilibrated IS_A boolean;

      P     IS_A pressure;
      T     IS_A temperature;
      vapor_options IS_A vapor_phase_options(ds,’Pitzer’,’Pitzer’);
      liquid_options IS_A liquid_phase_options(ds,’Rackett’,UNIFAC’);

      ds    IS_A td_component_data_set([’methanol’,’water’],’water’);
      feed_P IS_A pressure; feed_T IS_A temperature;
END test_vapor_liquid_flash;
```

We note several interesting points about this example.

- The errors ascribed to Ed are very much like those made by ourselves and the first users of model interfaces.
- Ed did not see any nonlinear equations.
- Ed saw quite a few logical and structural constraints in the WHERE statements of the model interfaces, but nearly all of them were satisfied quite naturally without direct attention from Ed. These constraints ensure that Ed is combining models in correct ways.
- Ed did have to look at the WHERE statements to find out the symbols for valid components and correlations. This suggests that an interface tool should search for and present these alternatives to the user.
- Ed has not used any operator except IS_A to construct this routine problem.
- Ed never looked at the internal content of the models. (He may get to that if the model does not converge.)
- Ed picks up quickly on looking one step ahead to define parts in his model. If he had not done so, the compiler would have continued to check for errors and generate messages about which passed objects and parametric values were missing each step of the way. Most of these checks can be performed at parse time, so Ed can find and fix his mistakes very quickly, i.e. in seconds rather than in minutes.
- Ed now has a flash model which can be solved for just the mass balances if he sets the value of `equilibrated` to **FALSE**.

We invite the reader to consider how long Ed would have worked to create his model if the libraries he used did not have interfaces, and whether or not Ed would have been able to meet all the configuration constraints listed in the **WHERE** portion of the model interfaces if these constraints went unwritten.

We do not see in this example two important properties of model interfaces. First, until all the objects passed to a model are constructed (and, in the case of constant data arguments, assigned) in a way which satisfies all the statements in the **WHERE** list of the model interface, the compiler will not attempt to construct the model. This ensures the orderly flow of shared subcomponents and valid parametric information from the root object into submodel objects. Orderly information flow is key to model source code transparency and to efficient model compilation.

Second, passed objects cannot have their types changed in the scope of models into which they are passed. Passed objects can be refined only in or above the application context where they are defined with **IS_A**. Assigning values to constants causes an anonymous type change, and **ARE_ALIKE**, **ARE_THE_SAME**, and **IS_REFINED_TO** may cause formal type changes. All these operations are disallowed on any part of a passed object. These restrictions ensure that an object passed into a submodel as a parameter does not get altered by the submodel, thus giving us better control over the deferred binding operations that create anonymous types. Passing an object through a model interface can cause no side-effects on the formal or anonymous type of the passed object. The model can, by using **ALIASES**, establish new names for subparts of a parameter.

5.6 LANGUAGE RESULTS

We now review the ASCEND IV language as we have defined and implemented it in the light of the design considerations and problems stated in Chapter 4. We aimed to create a language which has the properties of a good representation as outlined in Section 4.1. We shall see that not all the considerations and problems have been satisfactorily addressed in ASCEND IV.

We aimed to support casual users with little mathematical knowledge, engineering domain experts, modeling tool creators, and other computer programs. We believe that Code 5-12 and our scenario with Ed suggest that our language is quite suitable for delivering complex mathematical libraries to users who cannot, for whatever reason, afford to deal directly with the mathematical structure inside the models. Of course reaching the user's final objective, a solved or optimized model, requires that the libraries have well-coded methods attached to aid the user (or the solvers) in solving the resulting model. Domain experts can create novel models by writing additional constraints in a model constructed from standard library submodels. Domain experts can also take the library source codes and add new features, such as an alternative thermodynamic calculation method, easily because the flow of structural information in a parameterized library is made clear. We have not fully demonstrated that the language supports modeling tool creators, though as tool creators ourselves we rather like it. We have successfully constructed some examples of other computer programs which take models in our form and derive other representations. We will present these examples in Chapter 6. We now give a preliminary scoring of the language against the representation properties, in order to identify some remaining issues and to motivate some of the other tools we will discuss in Chapter 6.

Property a: We can now explicitly state the many configurational constraints that must be met when assembling complex mathematical models. As in Code 5-7 Line 4, we can now state explicitly how many copies of a repeated structure will be needed to construct a model in a way that helps the user (and the compiler) gauge the expense. We can state production models using only **IS_A**, **ALIASES**, and models with interfaces, eliminating the untraceable side-effects associated with deferred binding. We retain the ambiguous operators of ASCEND III to support workers modeling with incomplete or ambiguous information so that ASCEND IV can be used to help resolve ambiguous models and collect complete information.

Property b: We define a deliberately *incomplete* language specialized around stating nonlinear and logical equation-based models, contending that we should be very good at something in particular and anticipate that specialists in other areas will contribute the parts we choose not to investigate. We have added all the items identified as missing in ASCEND III (page 59) except two: the language cannot represent temporal logic and the language lacks the ability to define temporary local variables (stack data) in its methods. We also cannot yet represent partial derivatives explicitly in ASCEND IV, though [9] has shown at least one way that it might be done.

Property c: We demonstrated in Table 5-3 that the language facilitates storing data (compiling the model) rapidly provided an appropriate implementation is used. We can retrieve information from library models in source code form now because the new operators and parameterized types make it possible to write reusable libraries without applying the deferred binding operators to complex objects.

Property d: We can write transparent models, but the language does not force us to or stop us from doing so. Writing a transparent model in ASCEND requires making wise choices in the naming of variables and types. Even good names are sometimes insufficient to communicate all that might be needed, so we can use the **NOTES** defined in Section 5.4. There are still many questions that cannot be answered just looking at the model source code, however, such as “Where was the object associated with this name really defined in our application?” a question we frequently need to answer when debugging new models.

Property e: We have retained the conciseness of the ASCEND III set notation and object-oriented model structures in our definition of ASCEND IV.

Property f: Object-oriented languages help manage detail by encapsulating data, as ASCEND IV does. However, there are still many models which simply have too many equations, variables, set definitions, and submodels in any single layer (sometimes dozens) for us to browse them comfortably once they have been compiled.

Property g: The ASCEND IV syntax exposes many kinds of constraints, helping us to create complex models suitable for solution by mathematical programming algorithms and for translation into other languages.

Property h: The ASCEND IV language is computable by existing procedures written in ANSI C and Tcl/Tk that are distributed freely via the World Wide Web. The procedures are compilable on all the UNIX systems we have encountered. An upgrade of the Tcl/Tk sources to version 8 will make ASCEND compilable on personal computers running recent Microsoft and Macintosh operating systems[4].

Only through further testing of the ASCEND IV language and environment will we be able to ascertain the accuracy of our properties assessment. Some of the deficiencies noted in satisfying properties (a) through (h) are issues better addressed with tools that operate on the language or on the objects defined by the language. In the next chapter we present some tools which help address these deficiencies and other problems in object-oriented open form mathematical modeling.

4. This upgrade is in progress.

[1] Kirk Andre Abbott. *Very Large Scale Modeling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1996.

[2] Arne Stolbjerg Drud. “Interactions between nonlinear programming and modeling systems.” Working paper, ARKI Consulting and Development A/S, March 1997.

[3] Thomas Guthrie Epperly. “Implementation of an ASCEND interpreter.” Technical report, Engineering Design Research Center, Carnegie Mellon University, 1989.

[4] David Flanagan. *Java in A Nutshell*. O’Reilley and Associates, Inc., 1996.

[5] Brian Kernigan and Dennis Ritchie. *ANSI C*. Prentice, 1988.

[6] M. J. Maron. *Numerical Analysis*. Macmillan, 2nd edition, 1987.

[7] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[8] Bernt Nilsson. *Object-Oriented Modeling of Chemical Processes*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[9] M. Oh. *Modelling and Simulation of Combined Lumped and Distributed Parameter Systems*. PhD thesis, University of London, 1995.

[10] Eva Part-Enander, Anders Sjoberg, Bo Melin, and Pernilla Isaksson. *The MATLAB Handbook*. Addison-Wesley, 1996.

[11] Peter Colin Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[12] Vicente Rico-Ramirez, Benjamin Allan, and Arthur Westerberg. “Conditional modeling in ASCEND IV.” Technical Report EDRC/ICES TR 0-0-0, Engineering Design Research Center, Carnegie Mellon University, 1997. In preparation.

[13] Herb Simon. *Sciences of the Artificial*. MIT Press, 1978.

[14] Jacques Tiberghien. *The Pascal Handbook*. Sybex, 1981.

[15] Stephen Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley Publishing, 1988.

[16] Joseph J. Zaher. “Developing reusable model libraries in the ascend environment.” Technical Report EDRC TR 06-108-91, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA 15213-3890, July 1991. ASCEND III atoms,mathematics,components,thermodynamics libraries and the isom model.

CHAPTER 6 TOOLS TO SUPPORT MODELING WITH OBJECTS AND EQUATIONS

In this chapter we discuss a number of tools we have added to the ASCEND IV system, and specify several more that could be easily added. Most of the tools would be applicable in any object-oriented mathematical modeling system.

6.1 HELPING THE USER MANIPULATE MODEL INFORMATION

6.1.1 PARSER AND COMPILER MESSAGES

Most modern mathematical modeling languages have separate parsing and compiling stages, as we illustrated in Figure 3-4. We can substantially shorten each iteration in the process of writing new models if we can diagnose most modeling errors when the definition files are parsed, rather than waiting until the user attempts to compile an incorrect model. Because ASCEND uses a multipass compiler which does not execute statements in any well-defined order, we often have trouble deducing the ultimate cause of the error messages issued. Because there are often many objects made from a single definition, the number of redundant compilation error messages is also quite large. We save time because during parsing we handle each definition once and issue the diagnostic messages once. We handle each definition in a well-defined sequence, the order in which they are written in the input files. By issuing the messages once and in a proper sequence, we give the user the ability to start with the first error and work through the sequence correcting her errors. Fixing the first two or three errors is likely to shorten the list of remaining errors dramatically.

We have defined four classes of diagnostic messages: Style, Warning, Error, and Fatal. Style errors point out the use of syntax that may make models hard to reuse or hard to debug. At present every application of the ARE_ALIKE operator is diagnosed as poor style because the cause of a particular deferred binding on a particular object is very hard to trace when ARE_ALIKE is involved. Warnings point out statements that may not be correct but cannot be proven wrong due to the possibility of deferred binding. For example, relations are often written assuming that variables will exist in submodels, even though the formal type of the submodel is not sufficient to guarantee the existence of those variables. The parser cannot distinguish between this poor style and a misspelled variable name, so both cases are diagnosed. Well written reusable libraries will not produce Style or Warning diagnostics when they are loaded. Errors come from statements that the parser can prove are incorrect. Any definition which provokes Error diagnostics is rejected. For example, a model which declares a part using an undefined formal type is rejected. The Fatal diagnostics point out bugs in the ASCEND implementation. Users have suggested adding the ability to suppress both Style and Warning messages, but this has not yet been done.

6.1.2 REFINEMENT HIERARCHY

We have created a tool to display formal type hierarchies (class hierarchies) in ASCEND IV. This tool can help users figure out the relationships between model types, a task which is difficult when looking at a large set of library models. An example of a hierarchy from thermodynamic modeling[1] is given in Figure 6-1. At each node in the hierarchy we can ask the tool to display the complete list of statements which define the formal type. This

1. We make no claims for the merits of this particular classification of thermodynamic concepts.

list includes all the statements inherited from other definitions. At each node we can also ask for the list of formal types used to write the definition or for the list of variables the definition introduces. This tool helps make libraries more transparent (property (d)).

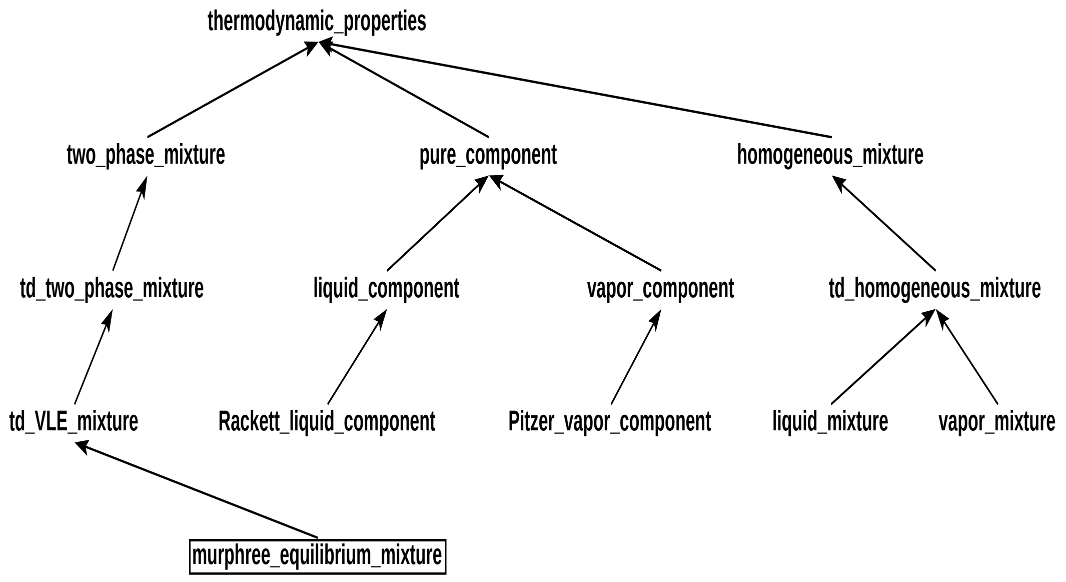


Figure 6-1 Library refinement ancestry (class hierarchy) display tool

6.1.3 INFORMATION HIDING

When browsing a large compiled object, we are easily overwhelmed by the amount of information available. We would like to suppress information that, for the moment, is uninteresting. Our definition of “interesting” is quite likely to change as we work with a model. In a model made from several hundred or more objects we cannot suppress information one object at a time: it would take us far too long. We have implemented general tools that allow the user to control which object types the object browser will hide.

For example, relations can be hidden. Sets can also be hidden, since the values of sets are frequently uninteresting once the relations have been compiled. This is not sufficient, however. Sometimes we want to hide information with a specific name only in specific kinds of context. For example, we might want to hide a large array, `trays,` within distillation column models so we can focus on just the input and output streams of our columns. We have implemented tools to allow this kind of suppression as well. At any time we can apply the tools to change what is hidden and what is displayed. These tools help the user manage detail (property (f)).

6.1.4 WHERE IS THIS CREATED?

When using our new parameterized modeling language, we often want to know where an object is created. **ARE_THE_SAME**, **ALIASES**, and parameter passing may create hundreds of new names for an object, but in a well-written model each complex object should only be created under one name. We can count the list of names under which an object was created in order to tell how many times it has been merged. We can also use the name under which an object was created to identify the locations where we are allowed to refine or merge it. We cannot refine or merge an object if we specify it with a name that was created by passing any object through a parameter list.

We have implemented a tool (the “Where created” button in our object browser) to identify the names under which an object is created. This tool helps make the language more transparent (property (d)). For example, we are browsing a distillation column, `flowsheet1.recovery_section.column2`, and find the object `td_data` which specifies the correlations and species to use in calculating physical properties. `td_data` is shared by every thermodynamic model object in the recovery section, possibly in the whole flowsheet, so it has hundreds or thousands of names. If we want to know where in our model we must go in order to modify `td_data`, looking at this list of names is not likely to be of much help. The “Where created” tool will give us a list with just one name for the object: `flowsheet1.recovery_section.component_data` because all the other names for the same object were defined by passing `component_data` as a parameter. We now know to look at the definition of the part `component_data` in the type definition of `recovery_section`.

6.1.5 REUSABLE DYNAMIC MODELING

We have created an initial value problem (IVP) solver tool which allows dynamic models to be built out of other dynamic models and to be integrated one part at a time or simultaneously. This tool makes it possible to create libraries of dynamic models, in contrast to ASCEND III where every dynamic model had to have several special models written for it to communicate with the IVP solver and those special models were not reusable. We have also implemented plotting tools to allow easy visualization of dynamic model output from one or several integrations using spreadsheet-like functions. These solving and plotting tools are described in [5]. These tools have been used extensively by other graduate students in our department.

6.1.6 MODEL REORDERING

Abbott [1] suggests an algorithm to derive good matrix reorderings based on part/whole hierarchies such as the ASCEND system provides. We have implemented variations of his algorithm in the ASCEND system and tools to let the user interactively select which reordering algorithms are applied to the Jacobian matrices used in ASCEND’s Newton-type solvers.

6.1.7 GAMS[2] CODE GENERATION

Building on unpublished stand-alone solver work of Joseph Zaher, we have worked with Chad Farschman, Vicente Rico-Ramirez, Mark Thomas, and Kenneth Tyner to create a GAMS code generator which allows users to create complex models in ASCEND and export them for solution with GAMS. This allows us to reuse ASCEND models with a variety of sophisticated solvers that we cannot justify spending research time to connect more directly to ASCEND. This also allows GAMS users access to an object-oriented modeling environment with many aids for debugging models, scaling equations and variables, and initializing complex models one piece at a time.

2. General Algebraic Modeling System [2]

6.2 IMMEDIATE EXTENSIONS BASED ON THIS WORK

6.2.8 CONSTRUCTED EXPLANATION OF NAMES

We could give the user a tool that takes a name and explains it using the **NOTES** available in ASCEND IV. The explanation of `fs.column4.tray[3].VLE.f` might be constructed using part names, model types, and **NOTES** made with the keyword `'is'` that we described at the end of Section 5.4. We see this in Example 6-1 where the text from `'is'` **NOTES** is shown in bold ***italics***.

> This real is of type `fugacity` inside model `VLE` which is of type `vapor_liquid_equilibrium` inside ***the condenser*** model `tray[1]` which is of type `equilibrium_tray` inside ***the oil separator*** model `column4` which is of type `radfrac` inside ***the Pasadena plant*** model `fs` which is of type `ethyl_alcohol_process`.

EXAMPLE 6-1 An application of **NOTES**

6.2.9 CUT AND PASTE MODELING

Given the simplicity of the mental tools needed during the creation of Ed's flash flowsheet in Section 5.5, (basically just reviewing parameter lists and satisfying explicit logical constraints) we could create a form-based or graphically-based modeling interface combining the information from model interfaces and **NOTES**. Creating such a graphical interface might be hard for some to justify as research, but we suggest that it is actually a challenging problem. We would like to create an interface tool which takes a library of models in a language such as ASCEND IV and uses algorithms that require no engineering domain knowledge to create an interface which makes sense to an engineering specialist. Many current industrial systems require handcrafted graphic user interface codes. Automating the production of these codes might be a significant step in lowering the cost of simulation software and might relieve expert modelers of the burden of designing clever interfaces.

6.2.10 CLASSIFICATION OF OBJECTS

As we have noted in Chapter 4, each complex object built from a reusable model has a formal and an anonymous type, where the anonymous type includes application specific set information such as the identity of chemical species, number of trays in a distillation column, and so forth. Given a completely compiled object, we could start at the outermost leaves of the object hierarchy and group objects by anonymous types, classifying our way toward the root of the hierarchy. If two objects are of identical anonymous types, then the second one could in theory share all the overhead of the first one except the memory locations required for variable data and relation residual data. In large flowsheet models we have seldom seen more than a few dozen anonymous model types.

Given this classification, we could automatically generate small pieces of object-oriented code in C, C++, or other suitable languages for the very fast evaluation of gradients. This approach would be much faster than generating code for each individual relation, a technique unsuitable for large models because the number of functions that most binary code compilers can handle efficiently enough to satisfy an interactive user is too small. Similarly, we could create a minimal set of mappings between our objects and data-oriented languages such as Express [3] or data translation tools such as CORBA [4].

Given this classification, we could also search for commonalities and differences between the names and anonymous types of parts in two models. Taking this sort of information, we could derive more abstract models, helping the user to classify her collection of models in ways that make it more reusable.

6.2.11 BEEFING UP ASCEND IV METHODS

We can build much more powerful methods for initializing models and carrying out other modeling activities if we extend the methods of ASCEND IV to allow the declaration of interfaces, local variables, and structures. For example we could write Code 6-1, where the function SOLVER connects to a piece of C code (possibly imported from a dynamically loaded binary library) capable of interpreting the data in SlvParameters to control a nonlinear solver. Thus we could write a column model with very sophisticated initialization strategies, making it much more reusable. All the ideas of “programming by contract” mentioned in Section 1.3.3 should go into the redefinition of ASCEND IV methods

Code 6-1
```
METHOD init_trays(status WILL_BE symbol);
      parameters IS_A SlvParameters(0.1,100,1e-8,'TearDrop2',2000);
      (* march from feed tray up to condenser, solving stagewise.*)
      FOR i IN [feed_stage..1] DECREASING DO
            RUN tray[i].reset;
            CALL SOLVER(tray[i], 'QRSlv', SlvParameters, status);
            IF status != 'OK' THEN
                  BREAK;
            END;
      END;
END init_trays;
```

6.3 FUTURE EXTENSIONS

There are a large number of other possibilities that are now explorable using ASCEND IV. The solving of general conditional models [7,6] represented with the **WHEN** syntax is an open problem. Developing a practical open form system which captures the essence of our work and temporal logic in a single reusable language seems to be a large challenge. Fully exploiting the semantics of ASCEND IV to improve the model compilation and the performance of solvers should prove to be a fruitful area of investigation. We hope that other interested researchers will pick up the tools we have created and use them in ways we have yet to imagine, sharing the results with the entire engineering community.

[1] Kirk A. Abbott, Benjamin A. Allan, and Arthur W. Westerberg. “Impact of preordering on solving the newton equations for process flowsheets.” *AIChE Journal*, 1997. In revision.

[2] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user’s guide, Release 2.25*. Scientific Press, 1992.

[3] Thomas R. Kramer, Katherine C. Morris, and David A. Sauder. “A structural EXPRESS editor.” Technical Report NISTIR 4903, National Institute of Science and Technology, Aug 1992.

[4] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, September 1995.

[5] Jennifer L. Perry and Benjamin A. Allan. “Design and use of dynamic modeling in ASCEND IV.” Technical Report EDRC 06-224-96, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1996.

[6] Vicente Rico-Ramirez, Benjamin Allan, and Arthur Westerberg. “Conditional modeling in ASCEND IV.” Technical Report EDRC/ICES TR 0-0-0, Engineering Design Research Center, Carnegie Mellon University, 1997. In preparation.

[7] Joseph J. Zaher. *Conditional Modeling*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, 1995.

CHAPTER 7 SUMMARY AND CONCLUSIONS

We began Chapter 1 with a list of six problems in process modeling. The problems cannot be solved individually, so we have taken a systemic view and presented a collection of ideas and tools which taken together can substantially improve how effectively we reuse modeling knowledge. In Table 7-1 we list each problem and which of the ideas most directly aid in solving it.

Table 7-1: Reusable modeling problems and solutions

	Problem	Our Solution
1	Varying user views	Open, hierarchical modeling.
2	Model evolution	Hierarchical modeling.
3	Communicating uses and limits	Built-in documentation. Modeling by contract.
4	Common modeling paradigm	Generalized, open equation-based models.
5	Tracking assumptions and data	Modeling by contract. System open to other information tools.
6	Negotiating modeling conflicts	Conditional modeling.

In Chapter 2, we have observed several recent developments in chemical process modeling, including movements toward both equation-based and component-based construction of process simulation models. We have focused on several reported difficulties with equation-based modeling, and we have argued that a new outlook on the structure of modeling systems and a new modeling language are needed to address these difficulties.

In Chapter 3, we have reviewed component software technology and visualized one possible world where software components can be used effectively to help solve engineering problems reliably and in a hurry. We have argued that complex modeling environments need to be built dynamically, possibly by borrowing pieces of other systems, and that real progress toward an efficient future requires mechanisms for a much broader information exchange than is supported by present commercial software. We have suggested that we should put the human in control of the system and that modeling systems should be designed to support interactions with other modeling systems in a peer-to-peer fashion.

We concluded Chapter 3 with an overview of the component-wise fashion in which ASCEND IV handles mathematical modeling problems to show the kinds of component behaviors that are already practical. We saw that our ASCEND IV solver programming interface design has been successfully built upon without fundamental changes by other graduate students to connect an advanced commercial solver (CONOPT), to handle new types of modeling information (conditional model logic representations), and to add improved component features such that a graphic user interface needs no built-in domain information about the solvers and models it must manipulate.

In Chapter 4, we have formulated goals for one of the *many* computer languages needed to support modeling complex engineered systems. We have reviewed several classes of mathematical modeling languages and systems to refine and expand our design goals and to justify our decision to create yet another modeling language.

In Chapter 5, we define interesting features of the ASCEND IV modeling language, a very general object-oriented mathematical modeling language suitable for:

- modeling by production modelers and expert modelers.
- defining both non-routine and very large routine models.
- developing modeling knowledge in an evolutionary fashion.
- managing large quantities of model information interactively.
- capturing the kinds of information required for automatically building user interfaces.
- investigating the construction of bridges among complex modeling tools, including users.

We have suggested that a general mathematical modeling language should be small and *incomplete*, providing general mechanisms for being used as part of larger systems and for using parts of other equally sophisticated systems. We believe such a language could provide support for all phases of the mathematical modeling process: model design, model construction, model solution, and model analysis. We defined parameter passing in a way that helps people who must reuse complex, open form models and, incidentally, helps the compiler build objects more efficiently. We defined **SELECT,** a mechanism to allow the compilation of alternative structures based on problem data. We defined **WHEN**, a mechanism to allow dynamic selection of model equations or structures before problem solution begins. We defined **ALIASES**, a mechanism to create new, possibly more meaningful, names for existing objects at virtually no compilation cost. We defined **NOTES**, a mechanism to support loose connections among different types of models, other tools, and users.

Our changes support a modeling style which is transparent, making it easier for a user to reuse, modify, and extend well-written open form models. We have used the new language to reimplement the ASCEND III libraries with much assistance from Jennifer Perry, an undergraduate in the Department of Chemical Engineering. After we constructed the flash model shown in Code 5-12, Perry[1] was able to create a new general purpose flash library in three[2] hours. This demonstrates that a well written ASCEND IV library can be built upon quite efficiently, even by someone with very little time, to solve more complex problems.

1. Perry has been instrumental in the development of ASCEND IIIc and ASCEND IV. We are deeply indebted to her for her many bug reports and other criticisms of ASCEND IV interface features and of our new language. It is supporting the work of Jennifer and other frustrated modelers that has motivated our definition of ASCEND IV.
2. We discovered a few easily corrected nonlinear equation errors upon testing the flash library, but all the complex model structures were created correctly the first time.

In Chapter 6, we have highlighted newly created and potential tools for helping modelers understand and use their models. These tools are discussed in the context of ASCEND IV, but they would be appropriate tools in any object-oriented system. We believe that the language and tools presented in this thesis indicate that robust, well-documented, portable, easy to extend and apply models, with good user interfaces could be produced quite rapidly using a language such as ASCEND IV. This could dramatically lower the cost of delivering high quality modeling capabilities to a wide range of users and facilitate communication among model users and model developers. However, both users and developers must be committed to exchanging a broad range of information in order to apply the methods we propose successfully.

There are many challenges presented in this thesis that remain to be met if we are to reach the ideal work environment presented in Chapter 3. We need a demonstration that a large, conceptually rich software system such as ASCEND can be decomposed into hundreds of independent tools that can be imported and used in other complex systems to solve modeling and mathematical problems. We need a demonstration that an information modeling system such as *n*-dim can be used to manage effectively several modeling systems comprising hundreds of complex, interdependent tools. We need the users and purveyors of process modeling software to agree upon open standards that make the creation of extensible, reliable modeling tools possible. Last, but possibly most important, we need a way for users to pay for software tools in proportion to the value derived from their use.