ASCEND: An Object-oriented Computer Environment for Modeling and Analysis. Part 1-The Modeling Language

by

P. Piela, T. Epperly, K. Westerberg, A. Westerberg

EDRC 06-88-90 C. 3

ASCEND:
An Object-oriented Computer Environment for Modeling and Analysis. Part 1- The Modeling Language

Peter C. Piela[1]
Thomas G. Epperly[2]
Karl M. Westerberg
Arthur W. Westerberg

Dept. of Chemical Engineering
and Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213

June, 1989

This research was funded by the Engineering Design Research Center, an NSF funded Engineering Research Center, under grant NSF CDR-8522616.

[1] Current address, Eastman Kodak Co., Rochester, NY 14650

Currently graduate student, Dept. of Chemical Engineering,University of Wisconsin, Madison, WI 53706

ABSTRACT

ASCEND (Advance System for Computations in ENgineering Design) is a new rapid model building environment for complex models comprising large sets of simultaneous nonlinear algebraic equations. In ASCEND the definition of a model is separated from the solving of it. This paper presents the ASCEND modeling language; a companion paper will describe the tools available to aid in debugging and solving models.

The ASCEND language is a type definition language that uses and extends object-oriented concepts, including refinement hierarchies, generalized arrays, part/whole modeling, partial and complete merging, deferred binding, and universal types. Dimensional consistency is required among all the equations.

This paper discusses the need for a modeling environment like ASCEND to aid the design process, comparing its functional requirements to earlier approaches. It then presents an informal definition of the syntax and associated semantics for the language. An example for solving mixed sets of ordinary differential and algebraic equations using a two point boundary value approach illustrates the modeling power of the language.

Table of Contents

List of Figures

1. Introduction

In chemical engineering process design, equation-oriented simulation has been shown to have significant advantages over the currently dominant sequential modular technology, yet it has not been widely adopted. We argue that the reason for this discrepancy is that the available tools are not good enough; they do not adequately support the abstraction and structuring concepts required to formulate and solve "real world" design problems.

Problems in engineering design can be usefully described as systems of mathematical relations. However, a designer working with this approach is forced to contend with many inherent complexities. Typically these systems are large, tightly coupled, and highly nonlinear; and can be extremely difficult to visualize. Furthermore, it is our view that equation-oriented modeling can never be entirely automated and that significant human intervention will always be required for such tasks as problem formulation and debugging. How well people can intervene will depend on the quality of tools available for these tasks.

With this view in mind we have begun the development of ASCEND, a computer environment for mathematical modeling based on "fourth generation" techniques. ASCEND consists of a model building language and a set of tools for working with the hyper-structures created by it. The language, motivated by the needs of chemical engineers but not tailored just to chemical engineers, has been developed with a focus on rapid problem formulation, modularity, physical correctness and the need to write generic models, instances of which can be safely modified to suit a particular application. The overall goal of this work is to bring the power of equation-oriented modeling to those working in engineering design, with significant support for problem formulation, examination, and debugging.

Many simulation and modeling systems limit the user to working with a predefined set of abstractions. These abstractions are built upon an underlying representation of equations and variables that are hidden from the user. Hiding severely limits the flexibility of the system. Should the user want to use an abstraction in a manner different from that for which it was written, it must be completely rewritten. Unlike these systems, ASCEND promotes the development and use of high level abstractions while allowing the user to work with the underlying structure when necessary. We believe that in order to maximize productivity and reusability of existing concepts the user should be able to work at whatever level of detail he or she perceives to be most appropriate.

Our work has confirmed that equational modeling is intolerant of errors in both formulation (e.g., writing redundant relations) and solution (e.g., identifying valid degrees of freedom), and thus a set of tools must be provided to support the modeling activity. The nature, scope, and performance of these tools is greatly affected by the way in which the mathematical systems are defined.

Based on these thoughts we have concluded that the design and implementation of tools for large scale mathematical modeling provide a rich area for research.

Whereas previous workers have focused on the problem of *solving* large systems of mathematical relations, our attention has been primarily directed to the problem of building and investigating these systems. Our hypothesis has been that a special-purpose modeling language will not only reduce the time it takes to build these systems, but will also give designers a formalism by which they can organize and share their work in a cooperative manner. Our goal has been to design a simple language which exhibits characteristics consistent with good language design in computer science, with special emphasis on simplifying the task of debugging, one which supports a style of use consistent with common design practice.

Based on our experience in chemical engineering, we have developed an understanding of the way in which designers build and solve models and how those models are structured. We use this understanding as a basis for the design of a new modeling language. We have selected examples from both chemical engineering and other domains, such as geometric reasoning and mathematics, to guide the development of the language. We define a new special-purpose modeling language. This language incorporates desirable features of existing computer languages such as static typing from Pascal and dynamic binding from object-oriented systems.

We have conceived and implemented tools for creating and operating on ASCEND data structures. These tools include a language interpreter and a solving engine. The development of both of these tools is described in separate documents (Eppeiiy, 1989) (Westerberg, 1989a) (Westerberg, 1989b). Although none of the basic solving techniques is new, the abstractions used in defining the software architecture and the way in which individual techniques are combined do not exist in other systems.

The flexibility of the language has been demonstrated in a number of ways. For example, the solving of differential equations is commonly treated as a different problem from the solving of algebraic systems. However, in practice, differential systems are solved numerically by making algebraic approximations. By writing some of these approximations in the ASCEND language we are now able to solve differential/algebraic problems in a single consistent framework (Smith, 1988). We shall illustrate the language later in this paper with such a problem.

ASCEND is a multi-tool environment operating on abstract data. By definition it is behaviorally complex, and thus a usable system can only be achieved *by augmenting the functionality of the tool with a model of the user.* In collaboration with a multidisciplinary team of designers, a project was initiated to create a highly resolved user interface for the ASCEND system, and in doing so, to identify issues for long term research. The originality of the user interface work derives from the intertwining of three separate requirements:

- That the system provide access to an amount of information which greatly exceeds the size and resolution of the display.
- That the system support the notion of independent tools operating simultaneously and in an arbitrary sequence on a shared database.
- That each tool have its own interface which must consistently reflect the current state of the database.
- That the system support graceful evolution. It should be possible to add new tools without having to redesign the interface. This requires a prototypical design with consistent appearance and functionality.

The ASCEND system has been implemented and is now being used. Problems have been solved in chemical engineering (Smith, 1988) and other domains (Woodbury, 1989). In an EDRC project several companies are testing ASCEND in-house, each for a two month period. This test will provide feed-back on it usability. The initial response from our users has been favorable; however, the ultimate worth of the underlying concepts of ASCEND can only be evaluated through extensive testing.

1.1. Background

ASCEND (Advanced System for Computations in ENgineering Design) is a computer environment for aiding the engineer in the analysis of designs which can ultimately be expressed in terms of mathematical relations. It contains tools for building, editing, and solving complex models, for storing problem descriptions, and for supporting flexible information exchange between the user and the system. To better understand where ASCEND fits into the overall process of design, we shall give a brief description of this process as we see it.

Design can be thought of as a process for creating something that satisfies a chosen set of goals. This process is often carried out in two distinct steps. In the first step, which we will call synthesis, the designer will use predominantly qualitative reasoning (experience) to postulate a conceptual design. The goal of the second step, which we will call analysis, is to build and solve a mathematical model of the design. The model will, in general, be a system of equations and variables with more variables than equations. The user must then select appropriate degrees of freedom and solve the remaining system for the parameters that characterize the design.

For example, given the problem of producing nitrogen from atmospheric air one might synthesize the following conceptual designs: low temperature distillation or molecular sieve adsorption. Analysis of these alternatives might involve determining the diameter of the distillation column or the required quantity of molecular sieve.

1.2. The fundamental assumption

This work assumes that mathematical relations are a good representation for a large number of engineering design/simulation problems spanning many disciplines. This representation allows us to decouple the declarative problem description from the procedural solution method, and suggests the following research question: What kind of tools will permit the user to build and modify mathematical models efficiently?

1.3. Motivation

This work is not aimed at improving the efficiency of solving design problems that are satisfactorily solved using existing technology. Our objective is the development of a next-generation design tool for tackling problems which are not currently attempted but are of interest to the community.

We began by asking the question, "Can significant reductions be made in the time taken to build and solve a model of several thousand equations?** At that time we believed that the answer was yes and we still do. Whereas much research in this area has focused on increased productivity through better solving techniques, our research concentrates on reducing the time taken to set up *correctly* specified problems. By correctly specified, we mean syntactically correct, topologically correct, dimensionally consistent, and structurally nonsingular.

Speed, however, is only one aspect of model building. Equally important is the profound way in which a

tool affects the way people think about design - how it causes them to structure and interpret problems, how it influences the building blocks they choose, and how those building blocks are then interrelated. We believe this link, between the character of the model-building tool and problem specification, makes an exciting area for further research. John N. Warfield made a similar observation in his monograph, "Structuring Complex Systems" (Warfield, 1974)

> It is reasonable to ask why, in view of the extensive literature on digraphs and matrices, there is any reason to be considering the apparently simple task of supplying entries to a matrix. Virtually all of the available literature takes for granted the existence of a matrix and proceeds from that point.
>
> There are a multiplicity of reasons for desiring effective methods of matrix development. Each of these individually might be considered a sufficient reason. When seen collectively, these reasons furnish a powerful rationale for developing such methods.

Warfield goes on to consider four of these reasons separately which collectively supply the rationale for tool development. These are: "The Tyranny of Numbers and Time," "The Difficulty in Being Consistent," "The Mixing of Physical and Social Elements," and "Difficulties in Interpretation."

1.4. A language for building models

Our problem is one of developing tools that will help designers in the formulation of mathematical models which are shared among many individuals, and solved by a computer. Our hypothesis is that a special purpose language could be a solution to this problem. The term "language" is used in the broadest sense - any set of symbols used in a uniform fashion by a number of people/machines who are thus able to communicate intelligently with one another. Although text is the natural means of expressing individual mathematical relations, this is not necessarily the case for groups of relations describing some physical object. For example, pictorial representations are the preferred means of communication in the case of a process flow or circuit diagram. We believe that future design environments will allow the designer to describe problems in a mixture of representations. In this research we concentrate on a textual language because at this stage it is more important to choose a good set of semantic primitives that can be expressed textually or graphically.

The problem of designing a language for building models has much in common with the development of an algorithmic programming language. From the extensive body of research relating to this field, we know that the transition from unstructured to high-level structured languages has greatly increased the productivity of software developers. The tools currently available for equational modeling are relatively unstructured, and best suited for solving small problems. These tools cannot be used in the formulation of large modular design problems which evolve over a period of time. We wonder if the transition from unstructured to structured could produce the same kind of gains in productivity for model-building that this transition produced in algorithmic programming languages.

It is important to note that, although the requirements of algorithm description and model building have much in common, there are some differences. Algorithm description is primarily procedural whereas model building is primarily declarative. In many algorithmic languages one defines types, which can be instantiated in multiple contexts. An instance is then modified by changing its type definition which in turn changes all other instances of that type. In design we also create instantiable prototypes; however, we desire to modify each instance in isolation, without changing the definition of the prototype.

1.5. Existing approaches

In the following section we will briefly mention some of what we perceive to be the good and bad points associated with current general-purpose modeling tools. The following list is not intended to be comprehensive, but representative of current technology.

Algorithmic programming languages: Languages such as Fortran are often used as supplements to existing analysis tools when first principles modeling is required. The major drawback with such languages is that they are designed for expressing algorithms and procedural notions. These goals are very different from those of the designer who wishes to describe structural relationships without conveying any information about the solving procedure. Because problem description and solution are intertwined, new code must be written for every solving scenario.

TKISolver: TKlSoWer is a software package for solving systems of non-linear equations. It is available on personal computers and is very popular for educational use. One might ask what it is about this package that makes it better for students than writing their own Fortran programs. The answer is that it is faster to use TKISolver, but why? TKISolver allows users to think in their own terms rather than those of the computer. Equations are input in a free-format nonprocedural manner; the system recognizes that physical quantities have associated units and the user is given the ability to manipulate degrees of freedom conveniently. A major problem with TKISolver which severely restricts the size of problem which can be attempted is that there is no mechanism for building complex models from simpler ones. In other words, the user is unable to represent a design as multiple layers of abstraction.

GAMS: GAMS (General Algebraic Modeling System) is a system built upon several popular mathematical programming packages such as MINOS (a non-linear optimization package). Great increases in productivity have been achieved with GAMS, primarily because it relieves the user of the burden of dealing with the rigid requirements on input format imposed by these packages. GAMS provides the user with a programming language in which complex models are constructed by indexing variables and relations over *sets* (enumerated types) which correspond exactly to the indices in an algebraic representation. Although the language provides a notation for concisely representing large systems of relations, there is little support for the specification of these systems in terms of meaningful parts. Users have tried to simulate part-whole relationships with multiple levels of indexing which is unnatural and results in overspecification which can be very inefficient. The GAMS language is primarily a notation. Variable names do not have a formal meaning when they are declared. The meaning is inferred from the relations which are written on the variables. An incorrect relation could therefore result in the wrong meaning.

Object-oriented programming languages: Examples of this new generation of languages are Smalltalk (Goldberg and Robson, 1983) and LOOPS. These languages were designed so that programs could be viewed as models of some aspects of the real world and that the dominant paradigm of programming would be simulation. These languages, although procedural, offer very powerful representational capabilities, permitting the designer to build complex models hierarchically. Notions of class and inheritance encourage the user to group similar objects according to common attributes and define specializations with added local detail. Concepts existing in previously written code can be reused in the definition of new ones. The major problem with these languages is that the object-oriented world is a separate one, containing many individual objects communicating via messages. Our experience in design suggests that problems are most naturally thought of as part-whole hierarchies in which a compound object is represented as a hierarchical decomposition of its parts. Parts are referenced by a pathname through the hierarchy. (The Unix file system is an example of a part-whole hierarchy.) Explicit representation of this notion is important in giving the designer the functionality that he or she requires. In object-oriented languages objects are grouped according to what they are (type), rather than to whom they belong.

ThingLab: ThingLab (Borning, 1979) is a graphic simulation laboratory which was designed and implemented as extensions to Smalltalk. The system is object-oriented and employs inheritance and part-whole hierarchies to describe the structure of a simulation. An interactive, graphic interface is provided that allows the user to view and edit a simulation. This work is notable for several major conceptual extensions to the Smalltalk programming environment which include: the addition of a part-whole hierarchy with explicit representation of shared substructure, the notion of constraints, and constraint satisfaction mechanisms. A principal part of the research was the design and implementation of a language that helps the user describe complex simulations easily. ASCEND borrows from this language the notions of part-whole representation and *merges*.

ThingLab was designed as an environment for constructing dynamic models of experiments in geometry and physics. In ThingLab constraints can apply to non-numeric objects such as text, as well as to numeric values. Constraints are satisfied by local propagation, and a relaxation method is employed only when a coupled subset of equations is discovered which must be solved simultaneously. Although local propagation is a reasonable approach for small problems that can be solved interactively, and can be completely precedence ordered, we believe it to be inefficient for solving large tightly coupled problems that are typical of engineering design. It is this class of large mathematical problems that motivates our research. In ASCEND a simulation comprises a set of simultaneous mathematical relations which is solved using numerical techniques. However, in order to take advantage of the improved efficiency associated with numerical techniques, problems must be well formulated. It is this requirement that has prompted our investigation into the role of language as a means of dealing with complexity and a mechanism for enforcing correct formulation.

In ThingLab a constraint is both a description and a command (it contains a rule for testing whether is is satisfied) and a set of messages which describe alternate ways in which it could be satisfied. In some cases the local satisfaction mechanism may require the user to define redundant constraints, that is constraints which would not be required if the entire problem were solved simultaneously using a numerical algorithm. Our approach has been to separate problem description and solution completely; a constraint is a statement of truth that does not convey any knowledge of how it is to be satisfied.

Working with the entire set of constraints as opposed to a subset has another advantage in that it allows us to employ algorithms which aid the user in correctly selecting degrees of freedom prior to solution (satisfaction). This is in keeping with our philosophy that only correctly formulated problems are submitted to the solver.

1.6. Are the existing approaches good enough?

Interactions with representatives of the chemical engineering design community suggest that the existing approaches are not good enough. It is our belief that there are a large number of problems which are either not attempted or are extremely time-consuming due to deficiencies in existing technology. Examples are:

- Analysis of novel designs, the building blocks of which are not contained in libraries belonging to existing simulators.
- Chemical processes with a large number of nested recycles.
- Problems in which inputs are to be calculated from specified outputs.

Most commercial simulators have been designed to perform optimally for certain classes of problem - those which involve standard unit operations and chemical species. In addition, there are often rigid constraints on how these unit operations can be used. For example, calculating outputs from inputs is straightforward; however, specifying an output and computing the required input is significantly more complicated. The penalty of this optimization comes when the user attempts to add new concepts. The following case study (Shah et aI, 1982), excerpted from work done at Alcoa, illustrates some of the difficulties associated with adding new unit operations to the ASPEN process simulator.

> Alcoa's main flowsheet oriented process is the alumina refining process (Bayer Process). In this process, solids are almost always present with fluid to affect the energy balance solutions. This eliminated most of the public version simulators from our consideration. ASPEN with solid handling capabilities, along with the capability to characterize nonconventional components like bauxite, was the logical choice. Our physical property correlations were not set up to be handled by a flowsheet simulator.... Another hurdle in our effort to model the Bayer process was the use of controls throughout the process. Even though other simulations have design specification capabilities, they restrict a control variable to be one of the process or block variables. ASPEN with Fortran capabilities, can handle combinations of process or block variables as control specifications. The Bayer process requires modeling of some unit operations (e.g. counter current decantation system) which were not provided by M.I.T. Being a table driven system, ASPEN requires adding a table entry to the system for **every** unit operation model to be added. It also requires writing a model interface to connect the equation solving model routine to the ASPEN executive routine. This not only demands excessive work, but also requires very good ASPEN knowledge. ASPEN compensated for this by providing a standard interface and table entry for the subroutines to be added by the user.

We believe that a more flexible design environment such as ASCEND could alleviate some of these types of problems.

1.7. The goals of this research

- To identify the kind of problems that designers want to solve.
- To identify a collection of key concepts that describe a methodology by which engineers build, solve, and manipulate complex design problems - based primarily on our experience in chemical engineering process design.
- To propose the design of a language which best supports this methodology and understand how it relates to existing approaches. In the design of the language we intend to strive for a minimal set of natural constructs.
- To develop a compiler for the language.
- To think about the design of the complete modeling environment built around the language (model libraries, solving engine, information transfer and human interface issues).
- To develop software that demonstrates these concepts.

1.8. Relation to other work

Virtually all chemical engineering process simulation systems are based on a sequential modular architecture. In this architecture a model of a process unit is a Fortran subroutine which will compute the output stream(s), given values for the input stream(s) and a sufficient number of additional equipment and/or operating parameters. These subroutines include not only the equations defining the unit, but also the method for solving them. It is becoming increasingly apparent, however, that this approach has several limitations, especially with problems where the flow of materials in the physical process does not coincide with the flow of information in the mathematical model. "Design" and "optimization" problems (Shacham et al, 1982) fall into this category. It has been suggested that these limitations may be overcome by adopting an equation-oriented approach in which the complete model of the process is expressed in the form of one large system of mathematical relations which are then solved simultaneously. There are a few commercially available equation-oriented systems (e.g. SPEED-UP, Prosys Tech Ltd.; TISFLO-II, Dutch State Mines; MASSBAL, SACDA) and several academically created systems (e.g. ASCEND-II, Carnegie-Mellon University; QUASILIN, Cambridge University, England; FLOWSIM University of Connecticut; SEQUEL, University of Illinois). To date the technology has not been proven by general use in industry; user feedback suggests that one reason for this is the difficulty associated with formulating large, well-posed problems.

The user community recognizes that efficient model building is an important issue. In an evaluation of SPEEDUP (Gupta et al, 1984) performed by Exxon, the ease with which new models could be developed was one of the evaluation criteria. In most of the equation-oriented systems listed above, models of new process units are written as Fortran subroutines which calculate the residuals for the equations describing the units and corresponding first derivatives. A notable exception is SPEEDUP, in which models of new process units are written in algebraic form in a special problem-oriented language. This language is a significant improvement over writing Fortran code. It has three levels of abstraction in specifying problem structure: equations and variables, models, and macros. However SPEEDUP does not have a generalized type structure, precluding the organization of types into inheritance structures (which minimizes duplication and allows for grouping similar objects) and type checking at all levels of abstraction (which minimizes debugging).

In their review of equation-oriented flowsheeting Shacham et. al. (Shacham et al, 1982) concluded that, for the approach to gain industrial popularity, technology must be developed in several areas, one of which is the verification of the input data. We believe that verification is strongly influenced by the form and method by which the data is input. Until now little work has been done on how the efficiency of the model building process might be improved.

While ^{H}dreaming^{M} about the capabilities of a future equation-oriented flowsheeting system, Westerberg and Benjamin (Westerberg and Benjamin, 1985) described certain attributes of a model building language. Some of these ideas were used as initial directions for this research.

For research in the area of problem description as it applies to simulation, we must look outside the domain of chemical engineering. In this summary we would like to focus on two extremely influential pieces of work. The first is CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions (Sussman and Steele, 1980); the second is ThingLab, a Constraint-Oriented Simulation Laboratory (Borning, 1979).

CONSTRAINTS is an interactive system organized around networks of constraints. A language exists for building hierarchical constraint networks. It supplies a set of primitive constraint types and a means of building compound constraint systems by combining instances of simpler ones. The constraint language was developed for representing knowledge about electrical circuits.

ThingLab is a graphic simulation laboratory developed in response to the question, "How can we design a computer-based environment for constructing interactive, graphic simulations of experiments in physics and geometry?" It should be noted that the underlying ThingLab system is not domain dependent. The design and implementation of the ThingLab specification language are extensions to Smalltalk. The system is object-oriented and employs inheritance and part-whole hierarchies to describe the structure of a simulation.

The ASCEND modeling language has several characteristics in common with both CONSTRAINTS and ThingLab. In all three of these systems, simulations are viewed as hierarchically organized networks of connected parts. Parts are instantiations of types which themselves contain instances of other types. Connection is achieved by merging complex connectors. Like ThingLab, ASCEND permits the user to extend previously written types through an inheritance relationship. This communality has been achieved by approaching the simulation problem from two radically different directions. Both CONSTRAINTS and ThingLab were primarily motivated by the requirements of interactive graphic simulation, whereas our research is based on the development of a strongly-typed yet flexible language which would guarantee the correctness of written models. We made an early decision not to try to implement ASCEND as extensions to a base language and thereby avoided constraints imposed by the design philosophy of the base language.

We believe that the feature that makes ASCEND unique is the flexibility that the user has in making changes to existing objects with the consequences of these changes propagating throughout an entire simulation. This is achieved without compromising the integrity of the existing problem description. We believe that this feature supports an important need of users who "design" by mutation.

At the system level, ASCEND is an example of a highly integrated environment comprising tools with shared access to a common database. These environments offer significant advantages over conventional technology in which collections of loosely coupled tools communicate with each other in a sequential manner. These advantages include better cooperation between tools; for example, the impact of one tool on the database is immediately seen by the others. In recent years there has been a growing interest in such environments with much research being done by both users of the tools and environment developers. A piece of research which is of particular relevance to our work was done by Garlan (Garlan, 1987). In his thesis Garlan develops an approach to the construction of integrated environments which he calls the view-oriented model. The essence of the approach is that each tool in an environment is defined in terms of a collection of *views* that prescribe appropriate representations of the data to be manipulated by that tool. Tool integration is then achieved by combining these views to produce collective descriptions of data shared among tools.

In the ASCEND language we define and manipulate declarative structures which are similar to views. For example, a simulation is ultimately a synthesis of the representation of an algebraic variable; we refer to this as ^{w}generic_real." Generic_real can be thought of as the -view" belonging to the solver.

2. A methodology for modeling and simulation

Most common programming languages are designed for expressing algorithms and procedural notions. These goals are very different from those of the designer who wishes to describe structural relationships without conveying any information about the solving procedure. An important part of this research has been identifying the key concepts of a methodology used by designers in the building of complex models. We believe that our methodology accurately represents current practice in the chemical industry and think that it is applicable to other design domains. The important concepts are described below:

1. Problem description should be separated from problem solution.
2. Mathematical relations should be written in a nonprocedural manner.
3. Certain activities associated with modeling are inherently procedural. One example is the assignment of initial values to variables, a precondition for solving.
4. Complex engineering models are built from simpler ones in a hierarchical fashion; e.g., a distillation column consists of sections, which each consist of stages, which in turn contain liquid and vapor streams, and so on.
5. Models are collections of connected parts - "almost-hierarchies" (Sussman and Steele, 1980).
6. All parts of a model should be accessible to a user of the model.
7. The reuse of previously written code should be maximized, so that existing concepts can be easily incorporated into the definition of new ones. For example, one might create a model called wilson_stream, which is a specialization of liquid_stream, which is a specialization of stream, etc. Wilson_stream inherits the structure of both stream and liquid_stream.
8. The designer should be able to define models which represent generic concepts.
9. The designer should not be burdened with having to define an endless number of prototypes. For example, a single "Ford Escort" prototype should suffice as the basis for creating a car with either standard or sports wheels.
10. The designer should be able to modify the structure of all parts of a simulation without compromising its integrity nor making alterations to previously written prototypes. The effects of valid modifications should propagate throughout the entire simulation. To do this we need a way of determining whether a desired modification is valid, and we need operators for making such modifications.

Items 4, 5, 6, and 7 (above) are consistent with an "expansionist" view of design, in which designs are part of larger designs, and the emphasis is redirected from the individual elements to a whole design with interrelated parts. Nadler (Nadler, 1985) says that

> Today, most design is carried out using a "reductionist" approach in which the object to be designed is decomposed into completely independent sections and subparts which are solved alone. Attempts to integrate these subparts usually lead to misfits. Since things are looked at separately, the interface area is usually neglected, and parameters generated in one subsystem may be contradictory with the same parameter generated in another subsystem.

We believe that domain independent tools like ASCEND, which allow for a structured and integrated approach to design, will aid in the production of better designs.

3. A description of the ASCEND modeling language

3.1. Introduction

ASCEND is a language for rapidly building complex mathematical models. Programming involves describing types which are composed of instances and relationships between them. Instances themselves are instantiations of other types.

3.2. A first example

The system of equations

$$f_1=2x_1^2+x_2^2-5=0$$

$$f_2=x_1+2x_2-3=0$$

is to be solved with the starting point x_1 • 1.5, x_2 = 1.0.

In the ASCEND way of thinking, all problems can be incorporated into the statement of larger problems. All problems are therefore defined as types, which can be instantiated in other contexts. In ASCEND structured types which contain relations are called MODELS. So let us begin encoding our problem into a MODEL, which we shall call "example." We begin model declaration with the keyword MODEL, which is then folbwed by the model's name. ASCEND statements can be entered in any order provided that any instance referenced within a statement has been previously declared. All ASCEND statements are separated by semicolons.

```
    MODEL example;
```

In order to write the equations f_1 and f_2 we must first declare the variables involved in these equations. This is done using the IS_A operator which associates one or more instances with a previously defined type. In this case it is the structured type "genericjeal," which is automatically included with any descriptions written in ASCEND. Generic_real represents an algebraic variable, and itself has a set of attributes which define its interaction with the solver.

```
        xl, x2 IS_A generic_real;
```

In ASCEND equations are written in a declarative style, and are treated as objects which can be optionally named by the user. In this example we assign the names f1 and f2 to our equations. The name associated with an equation has implicitly defined attributes which are discussed in the "language description" below.

```
        fl : 2*xlA2 + x2A2 - 5 - 0 ;
        f2 : xl + 2*x2 - 3 = 0 ;
```

In order to solve this problem, both x1 and x2 must be assigned initial values. Every instance of generic_real has a default value of 0.5, and hence the problem can be solved as is. However, as part of the problem description we were given a different starting point. These values are specified in an optional INITIALIZE section, which follows the declarative description. The INITIALIZE section contains assignment statements which are executed on request after the type has been instantiated.

```
    INITIALIZE
       xl:=1.5;
       x2:=1.0;
```

The MODEL is closed using the keyword END followed by the name of the model.

```
    END example;
```

The complete problem statement is given below, and is ready to be solved.

```
    MODEL example;
       xl, x2 IS_A generic_real;
       fl: 2*xlA2 + x2A2 - 5 - 0 ;
       f2: xl + 2*x2 - 3 - 0 ;
    INITIALIZE
       xl:=1.5;
       x2:=1.0;
    END example;
```

3.3. Basic concepts

The ASCEND analog of a program is called a simulation and is an instance of a model. Models are structured types built hierarchically from instances of other models, instances of atoms, and relationships between instances (figure 3-1). Atoms are primitive structured types used to represent physical quantities such as pressure, temperature or distance. Atoms and models are organized into inheritance hierarchies.

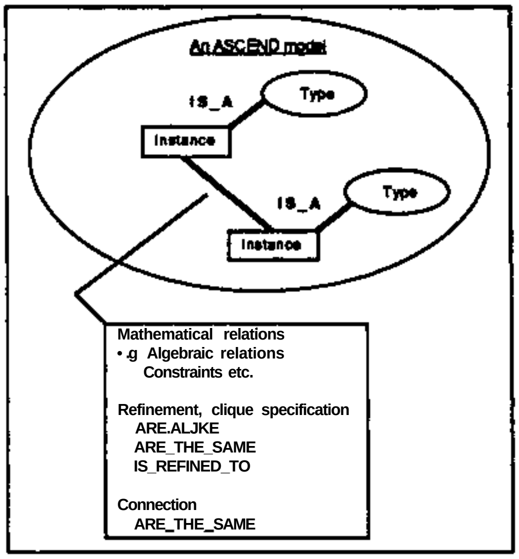

Figure 3-1: ASCEND models are networks of connected parts which are themselves instances of models.

In addition, there exist a number of elementary types that are basic to the language (e.g., real, integer, unit) and need not be declared.

There are two types of operators which specify what is done to instances. Relational operators (e.g., +, -) combine instances of atoms to produce algebraic equations and constraints. Language specific operators (e.g., IS_A, ARE_THE_SAME) allow the user to define and manipulate instances of atoms and models.

The following paragraphs introduce some basic language concepts.

Comments: Comments may be inserted between any two symbols. They are arbitrary character sequences enclosed in the comment brackets (* and *). Comments are skipped by the compiler and serve as additional information to the human reader. Nesting of comments is permitted.

Declarations: Every user-defined object must be introduced by a declaration, since declarations serve to specify certain properties of an object, such as whether it is a type or an instance.

Recursion: Type definitions cannot be recursively defined, that is a type definition cannot contain an instance of itself.[4]

Statements: Statements denote definition. There are two types of statements: elementary and structured. Elementary statements are not composed of any parts that are themselves statements. They are: the type declaration, instance declaration, array declaration, instance relationship, mathematical relation, and value assignment. Structured statements are composed of parts that are themselves statements. These are used to express conditional and repetitive execution. Language statements are separated by semicolons ";".

Local names: Local names are made up of letters and digits. The first character must be a letter. The underscore "_" counts as a letter. Local names are case insensitive, and can have a maximum length of thirty characters.

Global names: A global name refers unambiguously to an instance and is composed of a succession of local names separated by periods. It is a pathname through the instance hierarchy.

Example:

```
    column1.condenser.pressure
    my_car.door
    column1.stage[20].pressure
```

3.4. Type inheritance

Types are organized in simple inheritance (inclusion) hierarchies in which every interior type has a unique, immediate parent. See Fig. 3-2.

Base: The root node of an inheritance hierarchy is called the base.

Interior: All nodes except the base are termed interior.

Ancestry: The ancestry of a type is defined as the path between the base of its inheritance tree and

[4] It should be noted that although a type definition cannot contain an instance of itself an instance of the type can.

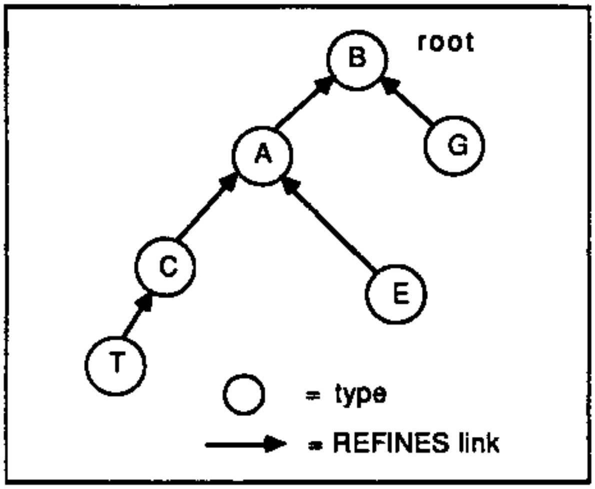

Figure 3-2: An example of a type inheritance tree,

itself. In Figure 3-2, for example, the ancestry of A is *B*<-A and the ancestry of **C** is *B<r-A<-C.*

Conformable ancestry: The ancestries of two types are said to be conformable if one is wholly contained in another. In Figure 3-2 the ancestry of G is *B<r-G* therefore A and C have conformable ancestries, whereas G and C do not.

Inheritance is strictly additive. An interior type inherits locally the complete declarative description of its parent, to which additional detail may be added. None of the inherited description can be deleted; this ensures that a type is a valid substitute for all its ancestors, because it contains the sum of all of ancestral structure.

REFINES link

Refines is used to add a type to an existing inheritance hierarchy. The statement

model a REFINES b;

means that model **a** locally inherits the complete declarative description of model **b.**

3.5. Declaring types

A type determines a set of values which instances of that type may assume, and it associates a name with the type. In the case of structured types, it also defines the structure of instances of this type. There are two different structures: atoms and models. The description of a structured type is divided into the following sections:

Type declaration: In the case of a model this section contains the model name and, if applicable, the name of the model it refines in an inheritance hierarchy.

In the case of an atom this section contains the name of the atom and the name of either another atom or an elementary type that it refines. If the atom is of base type REAL there may also be an explicit statement of its dimensionality. Both atoms and models can be declared UNIVERSAL.

Declarative description: In the case of a model this section contains all instance declarations, merges, refinements, mathematical relations, default value assignments for atoms of base type real, boolean, and unit, and fixed value assignments for atoms of base type integer and string.

In the case of an atom this section contains all instance declarations and default value assignments. These instances must be of elementary type and cannot be refined either within the atom definition or in a model in which the atom is instantiated. Merges and mathematical relations are not permitted.

Initial value assignment: This section contains a set of procedural operations describing how to assign initial numeric values to instances of base type real contained in the type.

Type completion: A type definition is completed with the keyword END followed by the name of the type.

The delarative description must follow type declaration. The initial value assignment section is optional; however, it must follow the declarative description.

3.5.1. The elementary types

There are only a few basic data types in ASCEND:

String: A sequence of zero or more characters surrounded by single quotes. Strings are used in the specification of array index values. The values of all instances of base type string must be known at compile time.

Integer: An integer with a value greater or equal to zero[5]. Integers are used in the specification of array index values and in mathematical relations, however, relations involving only integers are not permitted.

***We did not envision a need for negative index values.**

Integers are dimensionless. The values of all instances of base type integer must be known at compile time.

Boolean: Either TRUE or FALSE.

Real: A floating point number. In addition to value, REAL has implicit notions of dimensionality and therefore units.

Unit: Units associated with a physical quantity. The system maintains a single set of unit specifications, the contents of which comprise a group of pre-defined units, one for each fundamental dimension, and additional units which the user defines in terms of those existing in the set. Values are identifiers enclosed in braces, e.g., {hour}.

3.5.2. Declaring ATOM types

An atom is a structured type and can be a refinement of either an existing atom or an elementary type. It has an implicitly declared value and consists of a fixed number of instances (arrays are not permitted) of possibly different types, restricted to the elementary types described above. These instances cannot be refined when the atom is instantiated in another context.[6]

A description of a temperature consists of several parts such as its value, default value, lower bound and upper bound. These three items can all be placed into a single atom such as:

```
ATOM temperature REFINES real
                 DIMENSION tmp DEFAULT 298 {K};
   lower_bound IS_A real;
   upper_bound IS_A real;
END temperature;
```

An atom declaration begins with the keyword ATOM followed by the name of the atom. The declaration is continued with the keyword REFINES followed by the name of the type, which the atom refines, which may be either an elementary type or an existing atom.

If the atom is of base type REAL its dimensionality must be known. In the case when the atom is declared as a refinement of another, dimensionality will be inherited from its parent. If the parent is "undimensioned" an explicit statement of dimensionality is permitted. In the case when an atom refines

[6] The primary reason for keeping atoms simple is to improve compiler efficiency.

an elementary type an explict statement of dimensionality must be made as a part of the declaration, as was done in the above example. The dimensionality of an atom can be specified in one of the following ways:

DIMENSION expression : The expression is composed of operands which are the five fundamental dimensions: mass(m), length(l), time(t), temperature(tmp), charge(c), and moles[7] (mole), and operators which are *, / and ^ (exponentiation). Operands can be exponentiated by non-integer indices; however, the index must be written as a ratio of integers enclosed in parentheses. Reciprocal expressions can be written either as expression^-1 or 1/expression.

```
    l/t/t, l/t^2, m/(l*t^2), l^(1/2),
    m*(l/t)^2, (l/t)^2/tmp
    1/tmp, 1/t
```

DIMENSION * : Unspecified dimension

DIMENSIONLESS : All fundamental dimensions raised to the zeroth power

The declaration is completed with the optional specification of a default value which follows the DEFAULT keyword. The value must have associated units which are consistent with the dimensionality of the atom.

3.5.3. Declaring MODEL types

A model is a structured type which may refine an existing model. It may consist of instances of existing atoms and models, mathematical relations, arrays of types, and relationships among the latter.

A model declaration begins with the keyword MODEL followed by the name of the model. A model can be defined as a refinement of another using the REFINES keyword followed by the name of the model to be refined.

Example:

```
    MODEL car REFINES vehicle;
```

[7] A mole is a certain number of molecules, atoms, electrons, or other specified types of particles. In the SI and cgs systems a mole has about 6.02 x 10^{23} molecules; this is called a *gram mole*. In the American engineering system a *pound mole* has 6 02 x 10^{23} x 454 molecules. Thus the pound mole and the gram mole represent two different quantities. The user is responsible for using a consistent definition.

3.5.4. Declaring UNIVERSAL types

Structured types may be declared UNIVERSAL. Within a simulation, all instances of a universal type are implicitly made ARE_THE_SAME. A type is made universal by placing the UNIVERSAL keyword in front of either ATOM or MODEL.

Example:

```
UNIVERSAL ATOM speed_offlight DIMENSION 1/t REFINES real;
UNIVERSAL MODEL molecular_weights;
```

Universality is inherited as a part of type refinement; in other words, all descendants of a universal type are automatically made universal.

3.6. Declaring instances

All instances must be declared prior to use. The purpose of the declaration is to specify a set of properties belonging to the instance by associating it with a type.

The association of instance with type is not permanent; additional properties may be added to the instance by changing its type using language operators described later. An instance declaration is of the form: *list of instances* IS_A *base type.*

Example:

```
x IS_A real;
my__car IS_A ford;
exchl IS_A shell_and__tube;
display__unit IS_A unit;
```

3.6.1. Declaring arrays

An array is an ordered collection of elements of the same base type. It can have any number of dimensions. An array declaration contains the array name, index type for each dimension, and element type. Index types must be of base type integer or string, and integer indexes have a lower bound of zero. All array index values must have been assigned prior to compilation.

Example:

```
val[integer] IS_A real;
```

```
x[string]  IS_A mole_fraction;

u[integer]  is_a unit;

matrix[integer][integer]  IS_A real;
```

In the following example "species" is an atom of base type string, and ternaryjnteraction is a tripley dimensioned array of base type real.

```
ternary_interaction[species][species][species]  IS_A real;
```

3.7. Declaring units

ASCEND has an internal set of units corresponding to the fundamental physical dimensions. These are:

dimension	**language identifier**	**common name**
mass	kg	kilograms
length	m	meters
time	sec	seconds
temperature	k	kelvin
charge	col	coulombs
mole	mole	mole

The user can define new units in terms of existing ones. Units may be defined in multiple UNITS blocks.

A UNITS block begins with the keyword UNITS, which is followed by one or more assignment statements separated by semicolons each of which defines a new unit. The syntax of an assignment statement is: *newjanit* := *expression.* The expression is composed of operands which are existing unit identifiers and dimensionless numbers, and operators which are V, and ^{A} (exponentiation). Operands can be exponentiated by non-integer indices; however, the index must be written as a ratio of integers enclosed in parentheses. A UNITS block is closed with the keyword END.

Example:

```
UNITS
   lb := kg/2.42;
   newton := kg*m/sec/sec;
   joule := newton*m;
END;
```

3.8. Integer ranges

A range of integer values can be specified by writing a lower bound followed by the range separator "..^{M} and an upper bound. If the lower bound is greater than the upper bound, the range is interpreted as being empty. The bounds can be specified as integer expressions. Ranges can be used to create lists corresponding to parts of arrays. For example, the reference a[i..j] is equivalent to the list a[i], a[i+1]. . . .aU], given that i ≤ j.

Example:

```
1..10
1. .n_items
```

In a situation where the user requires a set of integers between two limiting values but does not know which limit is the larger, the following specification should be made:

```
limit1..limit2,  limit2..limit1
step[l. .n__step]  are_alike;
for i  :  i+5..j+6
```

3.9. Elementary statements

3.9.1. Merging and refining instances

Clique: **A** clique is a group of instances which have been declared ARE_ALIKE or ARE__THE_SAME either explicitly or transitively. The statement

```
a,  b  ARE_ALIKE;
```

creates a clique with members **a** and b. The further statement

```
b,  c  ARE_ALIKE;
```

adds the instance c to the clique.

Transitivity: The operators ARE_ALIKE and ARE_THE_SAME are transitive. The statements

```
a, b ARE_ALIKE;
b, c ARE_ALIKE;
```

imply

```
a, b, c ARE_ALIKE;
```

Instance refinement: Instance **A** is considered a refinement of **B** if the types associated with **A** and **B** have conformable ancestries and the ancestry of the type associated with **A** is longer than that of **B**.

IS_REFINED_TO operator

IS_REFINED_TO changes the type associated with a previously declared instance. The pre- and post-refinement types must have conformable ancestries with the latter having a longer ancestry than the former (figure 3-3).

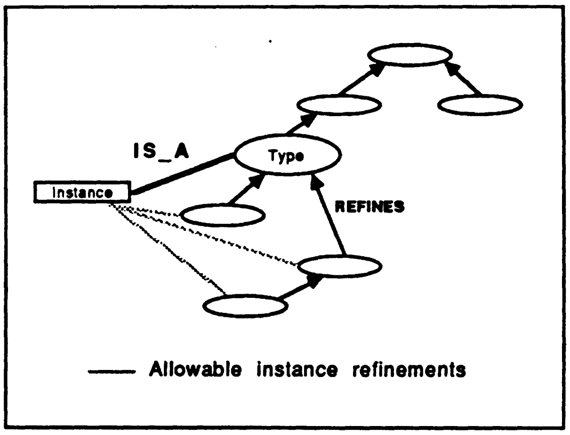

Figure 3-3: Instance refinement is the process of changing the type associated with a previously declared instance. The validity of a refinement is dependent upon user-defined inheritance trees. We require that the pre- and post-refinement types have conformable ancestries.

Example:

```
my_transport IS_A bicycle;
my_transport IS_REFINED_TO mountain_bike;
```

ARE_ALIKE operator

ARE_ALIKE coerces all members of a clique to the type associated with the most refined instance.

Instances declared ARE_ALIKE must be bound to types with conformable ancestry when the statement is made.

Example: See Fig. 3-2.

```
x1  IS_A  A;
x2  IS_A  B;
```

The intention of the following statement is to specify that x1 and x2 will be permanently bound to the same type, that is, any subsequent refinement of x1 will result in the same refinement of x2, and vice-versa. This is accomplished using the ARE_ALIKE statement which makes x1 and x2 members of the same clique. Their type then becomes the type currently associated with the clique which in this case is A because it has a longer ancestry than B.

```
x1,  x2  ARE_ALIKE;
```

ARE_THE_SAME operator

ARE_THE__SAME creates a clique in the same way as ARE__ALIKE, but goes further by merging the operands into a single most-refined structure.

Example:

```
x1  IS_A  A;
x2  IS_A  B;
```

The intention of the following statement is to merge x1 and x2 into a single instance. This is accomplished using the ARE_THE_SAME operator which creates a clique with members x1 and x2. In this case, x1 and x2 share a common instance structure in which the type of each instance in that structure is determined by looking at the type of the corresponding instance in both x1 and x2, and selecting the type with the longest ancestry.

```
x1,  x2  ARE_THE_SAME;
```

Note:

1. Although the current implementation does not allow the user to explicitly ARE_THE_SAME or ARE_ALIKE complete arrays, these actions are performed as a part of merging instances which contain arrays. The merged array will contain elements with a set of index values which is the union of the sets for each of the arrays to be merged. Given the array x with elements x[1] and x[2], and the array y with elements y[3] and y[4], merging would result in a single array with four elements which can be referenced by x[1], y[t], x[2], y[2], x[3], y[3], x[4],y[4].

3.9.2. Mathematical relations

The following paragraphs describe operators used in the definition of mathematical relations:

Arithmetic operators

The binary arithmetic operators are +, -, *, / and $^{\text{A}}$. There is a unary -, but no unary +. These operators are applied to instances of base type real. The operators +, -, and * are applied to instances of base type integer.

Relational operators

The relational operators are

=	equal
<	less
<=	less or equal
>	greater
>=	greater or equal

These operators are applied to instances of base type real.

Mathematical expressions

An algebraic expression consists of operands which are instances of base type real and integer, and arithmetic operators. Parentheses may be used to express specific associations of operators and operands. Each term in an expression must have the same dimensions.

Example:

```
x * <y+z) / p;
unit1.pressure - pipe.delp;
y[i-1] + x[i];
```

Defining a mathematical relation

A mathematical relation comprises two expressions which must have the same dimensionality separated by a relational operator. When the operator is "«" the relation is called an equation or an equality constraint. Relations involving other relational operators are called inequality constraints.

Relations can be assigned names which may be indexed when the relation is defined within a FOR construct. The syntax for defining a named relation is: *name* : *relation.* The name given to a relation has the following implicitly defined attributes:

included: A boolean variable which indicates whether the relation is included within the problem structure. The default value of TRUE specifies that the relation is to be included in the overall problem structure. Not including a relation is considered to be an instance refinement.

Example:

```
x + y = a + b;

FOR i:l..10 CREATE
   sum_j5art[i] : y[i] - y[i-l] + x[i];
END;

rad_flux :
   wb = 5.67e-5{erg/(cmA2*sec*kelvinA4)} * texnpA4;

TJiigh < 200{kelvin};

altl : x + y = z;
alt2 : x + y = t;
alt2.included :«• false;
```

3.9.3. Defining an objective function

An objective function can only be declared within a model and is done so by specifying an optional name and":" separator, followed by either the MAXIMIZE or MINIMIZE keyword and a mathematical expression to which the previous operator is to be applied. Like relations, the name associated with an objective function has the implicitly defined attribute "included," the function of which is described above. Only one objective function can be included in a simulation at any one time.

Example:

```
cost : minimize flowsheet.cost;

minimize column.rebolier_duty + column.condenser.duty;

maximize column.top_jproduct.x['methane'];
```

3.9.4. Associating units with a value

A "unit specification" is an algebraic expression opened by the symbol {and closed by the symbol}. The operands are identifiers corresponding to physical units, and the operators are \ /, \ A set of common identifiers is automatically loaded into the system, and is defined in the base insert file.

Example:

```
{m^2}
{ft/sec/sec}
{newton*m}
```

3.9.5. Assigning values to atomic instances

For an instance of base type real, boolean, or unit, assignment serves to replace a previous value by a new value. For instances of base type integer or string, assignment serves to permanently set the value of the instance. An attempt to overwrite a previously assigned integer or string value is considered an error.

The assignment operator is written as ":=". The type of the atom must be assignment-compatible with the type of the value.

When a value assignment is made to an atom of base type real, the user is expected to make an explicit statement about the physical units associated with the value. This is done by appending a "unit specification" to the value. If the units expression is omitted the atom is assumed to be dimensionless.

Example:

```
column1.pressure := 100{bar};
column1.pressure.fixed := true;
name := 'fred';
value[1][1] := 4;
column1.pressure.display_unit := {lbm};
```

3.10. Structured statements

3.10.1. The FOR statement

The FOR construct indicates that a group of statements will be repeatedly interpreted while a list of values is assigned to a variable. This variable is called the control variable of the FOR construct. The sequence of values assigned to the control variable is described by a list of "for-list-elements." These must be of base type integer or string. The scope of the control variable is confined to the FOR construct, so no formal declaration (using IS_A) is required. Its base type is that of the for-list-elements. It is not an instance, and therefore cannot be an operand of an ARE_THE_SAME or ARE_ALIKE relation; it can however be used within mathematical relations.

Example:

```
FOR i : species[1..nc] CREATE
   y[i] - k[i] * x[i];
END;

FOR i : 5, 4, 8,9 CREATE
   n[i] - a[i] + b[i];
END;

FOR j : L.nsteps CREATE
   x[j] = x[l] + (j-1) * step;
END;

FOR i : 'n2', 'ar', 'o2' CREATE
   y[i] - k[i] * x[i];
END;

FOR i : 1,2,8..10,20,tag[l..5] CREATE
   a[i] - b[i] + c[i-l];
END;
```

3.11. Assigning initial values to atomic instances

An optional initialization section may be included in the definition of a structured type. The beginning of the section is signified by the INITIALIZE keyword and ended by the type completion. Statements can either be value assignments or more complex actions written in the PASCAL-like initialization language in which variables are referenced by their ASCEND qualified names. Statements are interpreted in the order in which they are written. Initialization sections are inherited as part of type refinement[8]. The Pascal-like language has not yet been implemented; however, statements of the form *instance > numeric value* can be made.

Example:

```
MODEL column;
   .
   .
   .
INITIALIZE
   top_T := 380{kelvin};
   bottom_T := 500{kelvin};
   .
   .
END column;
```

[8] The current implementation does not support inheritance of initialization statements.

3.12. Compiler directives

3.12.1. %include "pathname[19]

The %include directive is used to read in a file ('pathname') containing descriptions written in the ASCEND language. This file is called an include file. The compiler inserts the file where the %include directive is placed.

ASCEND permits the nesting of include files, that is, an include file can itself contain an %include directive.

Example:

```
    %include  "/ascend/library/flowsheet_atoms.ascW
```

4. Language example

This section contains an example written in the current syntax of the ASCEND modeling language. It is intended to illustrate the modeling power of ASCEND.

Our goal is to develop a framework within which a future modeler can set up quickly a model to solve a set of one or more ordinary differential equations, possibly coupled with algebraic constraints, using a two-point boundary value formulation. To compare the effort, assume we could also prepare such a framework in Fortran.

The framework developed here will itself be an ASCEND model which we shall put into a model library for others to use. This model illustrates the use of generalized lists, refinement, part/whole concepts and the merge and deferred binding operations available fn ASCEND. We give it the name TPBV (two-point boundary-value) so we can be refer to it easily in the following.

In Fortran we would write a subroutine to be called by the user with a set of defined calling parameters. We would ask the user to call this subroutine and to pass into it the names of a user written subroutine to evaluate the time derivatives (right-hand-sides) of the ODE's and to evaluate the residuals of the equality constraints given values for the state and algebraic variables. We might also ask that this user supplied subroutine pass back the appropriate derivative information to our ODE solver so it can solve the two-point boundary-value problem using a Newton-based method. Our ODE solving subroutine would require many pages of Fortran code.

It will help to refer to the ASCEND model in Figs. 4-1 to 4-7 when reading the following descriptive material. To develop TPBV as an ASCEND model, we first need an interface into our model which gives the user a way to supply the ODE's and the algebraic equations. We do this by writing a model called **function_evaluation** in which we provide space for the user to store the values of the *njtate* state variables *y* and their derivatives with respect to time, *yjprime* (Fig. 4-1). We do not need to provide space in TPBV to pass along the algebraic equations nor their derivatives as ASCEND will handle these directly where they are written, which is different from the way our Fortran package would. We require the user to refine this model as the method to tie his/her actual problem definition into our IPBV framework.

Next we model how an ODE solver would take a single step. If we were to use the trapezoidal rule to integrate (modified Euler's method) for one time step, the values of the state variables at the beginning and the end of a single step are related by an equation of the form

$$W f l - W^{1} + W2) \text{ x } (yjrime\hat{}Vl + yj>rime_{end}[i])$$

For the modified Euler's method we create two function_evaluation models called fe[1] and fe[2], one the beginning of the step and one at the end.

In the spirit of ASCEND we create a model called **single_step** (Fig. 4-1) which can then be refined to be an Euler's method, a modified Eulers method and any other methods desired. The model single_step creates *nje* function_evaluation models and defines the overall stepsize *h.* We want all the models used within a single step for function evaluations to be of the same type so we set them all *alike.* This one statement is the secret behind the user supplied model being used at every point in the overall integration framework.

Finally Fig. 4-1 we form an array of single steps in the model **multistep.** We give the first step the alternate name *init_state.* We also merge the last function evaluation of step i with the first function evaluation of step i+1 with an *ARE__THE_SAME*statement. This one statement puts the single step models together toffof an array of successive steps.

form

The next models are **euler** and **mod_euler** (Fig. 4-2) and each is a refinement of single_step. Each inherits everything in single_step plus it sets the value for the number of function_eva!uation models to be 2 and defines the equations for the method.

We show also a fourth-order Runge-Kutta model called **Runge_Kutta_4** in Fig 4-3. Here a single step requires that we evaluate *yj>rime[i* at each of four points and then average them to predict the value of

```
MODEL function_evaluation;

        n_states                        IS_A    integer;
        y[integer],y_prime[integer],t   IS_A    generic_real;

END function_evaluation;

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

MODEL single_step;

        n_fe                            IS_A    integer;
        fe[integer]                     IS_A    function_evaluation;
        fe[1..n_fe]                     ARE_ALIKE;

        h                               IS_A    generic_real;
        h = fe[n_fe].t - fe[1].t;

END single_step;

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

MODEL multistep;

        n_steps                         IS_A    integer;
        steps[integer]                  IS_A    single_step;
        steps[1..n_steps]               ARE_ALIKE;

        init_state                      IS_A    function_evaluation;
        init_state,steps[1].fe[1]       ARE_THE_SAME;

        FOR i:2..n_steps CREATE
                steps[i-1].fe[steps[i-1].n_fe],
                steps[i].fe[1]          ARE_THE_SAME;
        END;

END multistep;
```

Figure 4-1: Basic Framework for Solving Two Point Boundary Value Problem

y[*i*] at the end of the step. We provide five function_evaluation models, fe[1] to fe[5], one for each of the four points and one for the values predicted at the end of the step.

We put the four constants *a* through *d* for this Runge-Kutta method into a *UNIVERSAL* model so that all copies of them will be *merged* into a single copy by ASCEND. The model for a fourth order Runge Kutta method follows, as a refinement of single_step.

```
MODEL eulers REFINES single_step;

        n_fe                            :=     2;

        FOR i: 1. . £e [1] . n__states CREATE
                fe[2].y[i]
                  = fe[l].y[i] + h*fe[l] ,yjprixne[i];
        END;

END eulers;

(*  ************************************************************  *j

MODEL znod__eulers REFINES single__step;

        n_fe                            :=     2;

        FOR i:1..fe[1].restates CREATE
                fe[2].y[i]
                  = fe[l].y[i] + h*(fe[l].y_prime[i]+fe[2].y_j>rixne[i])/2;
        END;

END mod_eulers;
```

Figure 4-2: Euler Refinements of Single_Step

For convenience we form a refinement of the multistep model for the case when the step sizes are all to be equal, writing the model **equal_multistep** (see Fig. 4-5). In the equal_multistep model we set the default value for the flag that indicates whether the step size *h* is to be fixed or calculated to indicate it is generally assumed to be fixed. As a default value, it can be overwritten later If desired by a model that has this model as a part or by the user through the user interface to ASCEND.

These models complete the coding of the framework for TPBV. We next need to test it, which we do with the models in Figs. 4-6 and 4-7. We model the draining of a tank, where the flow out is proportional to the square root of the height of the liquid in the tank. Assuming the tank has a constant cross-sectional area. the height is proportional to the volume of the liquid in the tank. We define several types of variables and give them dimensionality: volume, volumetric flowrate, length, area and time. ATOM definitions are typically already available from a library created when writing earlier models. These definitions in fact were taken from an existing library and did not have to written for this example.

```
UNIVERSAL MODEL Runge_Kutta_4_constants;

        a,b,c,d                              IS_A     real;
        a                                    :=       0.207106781;
        b                                    :=       0.292893218;
        c                                    :=       -0.707106781;
        d                                    :=       1.707106781;

END Runge_Kutta__4__constants;

 (* ------------------------------------------------------------------ *)

MODEL Runge_Kutta_4 REFINES single_step;

        n_fe                                  :=     5;
        abed                                 IS_A    Runge_Kutta_4_constants;

        fe[2].t,  fe[3].t                    ARE_THE_SAME;
        fe[2].t = fe[l].t + h/2;

        FOR i:1..fe[1].n_states CREATE
            fe[2].y[i]        = fe[1].y[i]  +  (h/2)*fe[l].y_j>rixne[i];
            fe[3].y[i]        = fe[1].y[i]  + h*(abcd.a*fe[l] .y_prime[ij +
                                      abcd.b*fe[2] .y_prixne[i]) ;
            fe[4].y[i]        = £e[l].y[i]  + h*(abcd.c*fe[2] .y^rixneCi] +
                                      abcd.d*fe[3] .y_prime[i]) ;
            fe[5].y[i]        = fe[1].y[i]  +  (h/6)*(fe[l].y-j>rixne[i]  +
                                    2*abcd.b*fe[2] .yjprime[i]  +
                                    2*abcd.d*fe[3] .y_prime[i]  +
                                    fe[4].y_prixne[i]);
        END;

END Runge_Kutta_4;
```

Figure 4-3: Explicit Fourth Order Runge-Kutta Refinement of Single_Step

Two models **are** needed once we have defined the atoms. The first, **tank_draining,** refines the model function_evaluation and is in Fig. 4-6. It supplies the physical relationships needed to relate the state variables, *y,* to their time derivatives, *yj>rime%* as one ordinary differential equation coupled with two algebraic equations.

The second model, **tank_draining__problem,** creates the overall problem.and is in Fig. 4-7. It includes as a part *ms,* an instance of the model equaLmultistep. it also defines the magnitude of the time step and

```
MODEL equal_multistep REFINES multistep;

        h                               IS_A    generic_real;
        h.fixed                         :=      true;

        FOR 1:1..n_steps CREATE
                steps[i].fe[l].t •  (i-l)*h;
        END;

        steps[n_steps].fe[steps[n_steps].n_f e].t
                  = n_steps*h;

END equal__multistep;
```

Figure 4-4: Refinement of Multistep for Equal Time Steps

makes ms.init__state into a tank_draining model by *deferred binding* (using an *IS_REFINED__TO* statement). Similarly it selects the integration method using deferred binding and shown here to be a Runge_Kutta_4. Both the tank draining model type and the method will propagate throughout the multistep model with these statements. Let us examine why for the tank draining model.

The init_state is the same as the fe[1] in the first step of ms. The ARE_ALIKE statement in themodel single_step makes all the function evaluations fe[1] to fefnje] for this first step alike; they all become tank__draining models when any one them becomes one, which happens when init_state is made into one. The merging with an ARE_THE_SAME statement of the last function evaluation model, fe[nje] of time step 1 to the first fe model of time step 2 upgrades the first function evaluation model fe[1] for the second time step into a tank draining model. fe[1] to fe[nje] for the second time step ARE_ALIKE; thus fe[2] to fe[n_fe] in step 2 also become tank draining models, etc, throughout the entire array of steps in the multistep model, ms. In a similar fashion, once one instance of the single.step models in MS is refined to be of a mod_euler type, all the others are also.

Many extension are possible for this problem. For example it could be extended by adding error estimation equations for the integration methods. These equations would add a few lines of code to each.

```
ATOM time REFINES generic_real DIMENSION t DEFAULT 1 {min};
        low_bound                       :=      0 {min};
        nominal                         :=      0.5 {min};
        display_unit                    :=      {min};
END time;

(* ----------------------------------------------------------- *)

ATOM volume REFINES generic_real DIMENSION 1^3 DEFAULT 100 {ft^3};
        low_bound                       :=      0 {ft^3};
        nominal                         :=      10.0 {ft^3};
        display_unit                    :=      {ft^3};
END volume;

(* ----------------------------------------------------------- *)

ATOM vol_flowrate REFINES generic_real DIMENSION 1^3/t
                                         DEFAULT 10 {gpm};
        nominal                         :=      10 {gpm};
        display_unit                    :=      {gpm};
END vol_flowrate;

(* ----------------------------------------------------------- *)

ATOM length REFINES generic_real DIMENSION 1 DEFAULT 10 {ft};

        nominal                         :=      10 {ft};
        display_unit                    :=      {ft};

END length;

(* ----------------------------------------------------------- *)

ATOM area REFINES generic_real DIMENSION 1^2 DEFAULT 30 {ft^2};

        low_bound                       :=      0 {ft^2};
        nominal                         :=      30 {ft^2};
        display_unit                    :=      {ft^2};

END area;
```

Figure 4-5: ATOM Definitions Needed for Problem. ATOM definitions are typically available in an existing library

For the model as written, the framework in Figs. 4-1 to 4-4 requires only sixty lines of code. It can solve mixed sets of ODEs and algebraic equations using a two-point boundary value approach. Included in these sixty statements are three different integration methods: eulers, mod_eulers and Runge_Kutta_4.

```
UNIVERSAL MODEL tank_draining_parameters;

        k                               IS _ A    generic_real;
        k.fixed                         :=        true;
        k                               :=        10.0;

        a                               IS_A      area;
        a.fixed                         :=        true;
        a                               :=        2 {ftA2};

END tankjdrainingjparameters;
(*-------------------------------------------------------------- *)

MODEL tank__draining REFINES function_evaluation;

        n_states                         :=       1;
        vol                             IS_A      volume;
        dvol_dt, F_out                  IS_A      vol_flowrate;
        ht                              IS_A      length;
        params                          IS_A      tank_draining_parameters;

        y[l],vol                        ARE_THE_SAME;
        yjprime[1],dvol_dt              ARE_THE_SAME;
        t                               IS_REFINED_TO time;

        dvol_dt =   - F_out;
        F_out = params.k*1.0 {gpm}*(ht/(1.0 {ft}))A(1/2);
        ht = vol/params.a;


END tankjdraining;
```

Figure 4-6: Tank Draining Model for Relating States *y* and Their Time Derivatives *yjprime*

We have also added an orthogonal collocation method, for example, but this method is not shown here.

5. Discussion

In this paper we have argued for the need of a modeling system such as ASCEND to aid engineers in the design process. We gave specific functional requirements for ASCEND and then described the syntax of the modeling language which resulted from our trying to meet these specifications. An example illustrates the language and, we hope, also illustrates the effectiveness of it to represent a moderately complex model in an easy to follow manner. The ASCEND language is a very different kind of language from those available previously for modeling complex systems. Our experience so far indicates that it does

```
MODEL  tank_dra±ning_problem;

        xns                             IS_A      equaljmultistep;
        xns.h                           :=        0.2  {rnixi};
        xns.n_steps                     :s        8;

        xns.init_state                  IS_REFINED_TO     tank_dra±ning;
        xns.steps[1]                    IS_REFINED_TO     Runge_Kutta_4;

        xns.init_state.y[l]             :=        10  {ft^A3};
        xns.±nit_state.y[1].fixed       :=        true;

END  tank__dra±n±ng_problem;
```

Figure 4-7: Overall Tank Draining Problem Model

support the rapid writing of complex models by those familiar with it. It is also becoming quite clear that there is a cost involved to learn the language as it requires a very different approach to modeling than is used by most practicing engineers. One has to learn to think differently about modeling. The language has only a few main features to it: refinement *(REFINES),* part/whole structuring *(IS__A)*$_t$ partial *(ARE_ALIKE)* and total merging *(AREJTHE_SAME)*$_\%$ deferred binding *(IS__REFINEDJTO)* and the notion of *UNIVERSAL* models. These, combined with the tools available in the user interface to debug and solve, allow ASCEND to be used in very subtle ways.

In a follow-up paper, we shall describe the activities involved and the supporting tools available in ASCEND for debugging and solving a model such as this. This companion paper will also describe the user interface. These tools and the interface to the user are as important as the language in making ASCEND into an effective system for building complex models quickly. We shall describe the debugging and solving activities only very briefly here.

Debugging facilities in ASCEND aid the user in correctly setting the degrees of freedom. If there are too many variables picked as being fixed (so there are m equations to solve for n<m variables), the system can suggest to the user a set of variables, one of which he/she should release to be calculated. Simply clicking the mouse on one of the variables in the list will convert it to one that is to be calculated. This process continues until the there are an equal number of equations and variables. If, on the other hand,

there are more variables to be calculated than equations, the system can give a list of variables which are eligible to be fixed. These tests are based on the structure of the equations only - i.e., based on noting which variables occur in which equations. In addition, the system can detect numerical singularity and tell the user which equations are currently not behaving independently. Such a behavior can indicate that the model has been written with an extra dependent or inconsistent equation.

ASCEND also checks the dimensional consistency of all the models when it instantiates a model as it is creating a simulation. A way to think of this is that every equation is really supplying two relations; one is among the values and the other is among the dimensions of the variables involved. Both are written in ASCEND as a model is instantiated, and the latter is used to infer dimensions not explicitly stated or to detect inconsistencies. Dimensional inconsistency is typically not detected in other modeling environments except by noting that strange numbers are being computed. Often even the numbers do not give it away, and one can be assured of dimensional correctness only by tediously executing the code by hand on an example. It is possible to model in ASCEND by declaring all variables to be of the type generic_real and bypass the dimensional checking. However, one then does not get this powerful debugging assistance which is provided. If one is serious about writing a correct model, he/she must take advantage of this feature.

Once a model has been created in ASCEND, any part of it can be selected and solved. In the next paper we shall show how one can use the INITIALIZE sections to aid in setting the flags to establish which variables are to be fixed and which are to be calculated *when a part is selected.* In the model of a rankine cycle which has been created using ASCEND, thecomplete model was debugged by first selecting a pure component stream within the model and executing only it. Since the stream has its physical property calculations as a part of it, this calculation was set up to compute the physical properties given its temperature and pressure. Once the physical property model was debugged, the compressor model was selected, then the heat exchanger model and finally the valve. After these parts were executing as expected, the overall cycle was selected and debugged. All of these runs were done with the creation of but one simulation - that for the overall model.

To argue qualitatively for the power of the ASCEND system as a modeling system, the fourth author of this paper (A. Westerberg) coded the model TPBV presented in the previous section (excluding the Runge_Kutta_4 model) in about an hour once an outline for the code was completed. It was typed in and debugged sufficiently using the tools available to get a first execution in less than one additional hour.

The power of using an equation solving approach is evident. When solving, the user is free to fix any set of variables desired to make the problem into one which is well posed. Thus, in our example tank draining problem, (1) the volume can be set at the time of 1.6 minutes instead of setting it at the initial time, or (2) the volume can be set at both the initial and final times while letting the step size be calculated (the same as computing the total time needed), or (3) the volume at the initial and final time plus the step size can be fixed, while computing the constant *k* (which can be thought of as corresponding to the opening of a valve in the drain). Obviously many other options are possible, as long as they lead to legitimate computations.

The computer times for the system are also of interest. We will report on these more in the second paper. It should be noted that both instantiation and solution times depend strongly on other attributes such as problem structure. In particular, a beginning user will generally write a first model that has little or no structure. It will be a single model with perhaps many equations but with no refinements and no part/whole (IS_A) statements involving model types and few merging statements. A two thousand equation model written in this form instantiated in about two minutes. On the other hand, a very highly structured 300 equation model could easily take the same time. Typical instantiation times range from less than a second for a two equation, single simple model example to over 30 minutes for a 2000 equation, 3000 variable example displaying considerable complexity. Execution times for these models are typically much less. The 2000 equation, 3000 variable model solved in 3 minutes. All of these times are on an Apollo DN4500 workstation.

It should be noted that with the merge statement, ASCEND models are very compact, and often the model size can be 50% to 100% larger if merges are replaced by equations of the form y=x.

There are two solvers in ASCEND. The first is a specially written solver using a modified Marquardt method. When asked to solve, the system precedence orders the equations and solves them one block of equations at a time, following that order. Using the Runge-Kutta explicit integration model above and setting the initial volume gives a set of equations that fully precedence order, i.e., they can be solved one at time in sequence. ASCEND will do just that. Derivatives needed for the Marquardt method are evaluated explicitly by the system. The second solver is MINOS-Augmented, a state-of-the-art optimization package available from Stanford University. It is invoked automatically anytime the model includes an objective function to be maximized or minimized.

References

Alan Borning. *THINGLAB - A Constraint-Oriented Simulation Laboratory.* PhD thesis, Dept. of Computer Science, Stanford University., 1979.

Thomas. G. Epperly. *Implementation of an ASCEND interpreter.* Technical Report, Engineering Design Research Center, Carnegie Mellon University, Jan 1989.

David Garian. *Views for Tools in Integrated Environments.* PhD thesis, Dept. of Computer Science Carnegie Mellon University, 1987.

Adele Goldberg and David Robson. *Smalltalk-80 : The Language and Its Implementation.* :Addison-Wesley, 1983.

P.K. Gupta., R.C.Lavoie, and R.R. Radcliffe. An Industry Evaluation of SPEEDUP . Paper presented at the AIChE 1984 Annual Meeting, San Francisco meeting of the AIChE. Nov 1984.

Gerald Nadler. Systems Methodology and Design. *IEEE Transactions on Systems, Man and Cybernetics,* 1985, *SMC-*75(6),.

M. Shacham, S. Macchietto, L.F. Stutzman and P. Babcock. REVIEW - Equation oriented approach to process flowsheeting. *Computers & Chemical Engineering,* 1982, 6(2), 79-95.

V.B. Shah, P. Gacka and J.M. Langa. *The application of ASPEN flowsheet simulator at Alcoa,* pages 56-65. AIChE, 1982.

Oliver Smith. *Solving Optimal Control Profiles as Algebraic Equations.* Technical Report, Engineering Design Research Center, Carnegie Mellon University, Dec 1988.

G.J. Sussman, and G.L. Steele Jr. CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence,* 1980, *14,*1-39.

John N. Warfield. *Structuring Complex Systems.* Technical Report, Battelle Memorial Institute, Apr 1974.

Karl Westerberg. *Development of software for solving systems of linear equations.* Technical Report, Engineering Design Research Center, Carnegie Mellon University, Jan 1983.

Karl Westerberg. *Development of software for solving systems of nonlinear equations.* Technical Report, Engineering Design Research Center, Carnegie Mellon University, Jan 1989.

A.W. Westerberg, and D.R. Benjamin. Thoughts on a future equation-oriented flowsheeting system. *Computers & Chemical Engineering,* 1985, 9(5), 517-526.

Robert F. Woodbury. Variations, Features and Prototypes in Solids: A Declarative Treatment.