Carnegie Mellon University
Research Showcase @ CMU

Department of Chemical Engineering | Carnegie Institute of Technology

1989

A future computer environment for preliminary design

Arthur W. Westerberg
Carnegie Mellon University

Carnegie Mellon University.Engineering Design Research Center.

Follow this and additional works at: http://repository.cmu.edu/cheme

This Technical Report is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Chemical Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

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

A Future Computer Environment for Preliminary Design

by

Arthur W. Westerberg, Peter C. Piela,
Eswaran Subrahmanian, Gregg W. Podnar, William Elm

EDRC 05-42-89
Carnegie Mellon University

A Future Computer Environment for Preliminary Design [1][2]

by

Arthur W. Westerberg
Peter C. Piela [3]
Dept. of Chemical Engineering
and Engineering Design Research Center
Carnegie Mellon University

Eswaran Subrahmanian
Gregg W. Podnar
Engineering Design Research Center
Carnegie Mellon University

William Elm
Westinghouse Electric Corporation

Pittsburgh, PA
April, 1989

Abstract

In this paper we consider the creation of future computer-based design environments to support a team-based preliminary design process. Adopting the view of Bucciarelli [1988], we see design as a social process among the team members. Each team member has his (or her) own view of the artifact to be designed, and each operates in his own "object world." We list a set of attributes we see as needed for a computer-based environment to support this process. After suggesting that the activity of design is a form of continual evolutionary model development by the participants, we argue that it can be captured by providing a support system to record and organize models as they are developed and shared. Based on our work with creating the modeling language ASCEND, we propose a number of specific concepts on which the design of such a support system can be based.

---

[1] Paper presented at Foundations of Computer Aided Process Design (FOCAPD) Conference, Snowmass, CO, July 9-14, 1989.

2 This work has been supported by the Engineering Design Research Center, an NSF Research Center.

[3] Current address: Eastman Kodak, Rochester, NY.

University Libraries
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Introduction

A few months back, when conference organizers extended an invitation to the first author above to prepare a paper for this conference, the title they proposed was The Future in Design and Design Environments.[11] When Jeff Siirola subsequently circulated abstracts for the invited presentations, it was clear that this presentation and that by Roger Sargent could seriously overlap. To avoid conflict, therefore, this paper will consider only the latter topic: future design environments. We shall discuss in this paper the creation of support systems to aid design teams to carry out the design process through several of its stages. It will be the organization of support systems to integrate people, information, tools, multiple representations, etc., that will be of concern here. Further we shall emphasize supporting design during its early stages, where the concepts are to be established on which the design is to be based. We shall not consider the development of individual tools to carry out only one of the several steps required, even if that tool is rather elegant.

With this emphasis, it seemed at first that this paper would be rather speculative. There are only limited efforts on creating design environments in chemical engineering. The Design Kit being created by George Stephanopoulos [Stephanopoulos, 1988] is an exception. Perhaps the most extensive literature on creating design environments is for the development of large computer software systems. This topic is known as Computer Assisted Software Engineering or CASE [Fisher, 1988]. There has also been considerable work in putting together tools to support the design of VLSI where the approach is toward full automation of the process [Daniell and Director, 1988]. Tools to aid in the discovery of the concepts on which to base the design are not emphasized.

This paper is based on a joint project between the Engineering Design Research Center (EDRC) at Carnegie Mellon University (CMU) and the Westinghouse Electric Corporation to develop a prototype design environment to aid the preliminary design process. To understand how to create an effective and comprehensive design environment has required us to define carefully what we think the design process is, to note just how varied it can be and to propose how we could create tools to support it.

Thanks are owed to coworkers on this project: Lorenz Biegler and Michael Rychener during the first year. John Gallagher of Westinghouse is especially thanked for his continued support,

input and interest in this project. Many of the ideas in the first part of this paper are in Subrahmanian et al [1989].

Design Automation vs Design Support

For certain classes of design problems, we can imagine that many of the steps can be fully automated. Indeed if the technology is well enough understood, it makes sense to strive for "pushbutton" design. The next conventional McDonalds restaurant can almost certainly be the result of a totally automatic design process. Where an artifact for which the design concepts as well as the methodology for carrying it out are unknown, it makes sense to talk of only aiding the design activity especially in its earlier stages. What should such aids be? This question is the basis for this paper.

Characteristics of the Preliminary Design Process

We shall begin by describing briefly the project between the Engineering Design Research Center and Westinghouse. During the first year the EDRC participants observed the bi- to tri-weekly meetings of a team from Westinghouse as they designed a new control system for coal-fired electric power generation plants. The control system was to be based on newly available technology and thus did not represent a routine design activity.

The design team came from several different parts of the company, some from about a hundred miles away. Team members were selected because each was thought to bring an expertise deemed necessary to carry out the project.

The EDRC observers found the team going through an initial step of information gathering. For the early meetings, each team member was assigned the job of looking among company reports, in textbooks, in the library, etc., for information on a specific aspect of the process thought to be relevant to the design. This information was shared by distributing copies of reference materials, handing out of memo reports, making formal presentations using overheads and by informal discussions, often using a whiteboard which could be copied by the push of a button. Each person involved became the owner of a pile several inches high of materials. While

we all knew roughly what was in this material, we had difficulty finding specific information as the information was neither indexed nor cross-referenced.

Quite early in the project the design team formulated desired attributes for the power generation process control system. They also sketched different control schemes, at first at a very abstract level of description, to meet these requirements. Much time was spent trying to understand what someone was really saying. Where holes were found to exist in the team's knowledge base, they would either go back into the information gathering mode or would invite experts from elsewhere in the company to attend the next meeting.

After observing this activity, the EDRC participants proposed different design aids which might be possible to create to make this design process significantly more effective. Our current joint project is to develop a prototype design environment to aid the preliminary design process. To create such an environment, we need to define more precisely what we mean by the preliminary design process; we highlight next two papers which provide useful insights into this process.

Design Is a Social Process [Bucclarelll, 1988]

Many models of the design process have been proposed. We have been quite taken by the model given by Bucciarelli. He notes first what he believes are insufficient models for the design process. For example, an Artificial Intelligence model views it as a knowledge base to be rapidly accessed, with the use of heuristics to generate and test alternatives. For many of those involved in teaching design, it is viewed as a control process moving from one step to the next with feedback loops which result in iteration of earlier steps. The steps include analysis of need, brainstormlng, generation of conceptual designs, evaluating these designs, etc. He states that feedback control in the wrong metaphor! It may work for teaching design, but design does not follow such a neatly packaged sequence (see also Rasmussen, 1988, for a discussion of the very "unordered" sequence of activities involved in the design process). Finally he points out that others view it as the management of process. Designing here is effective communication.

Bucciarelli argues that the design process is a *social process.* While some of the activities are solitary, many others are collaborative. Attending review meetings, celebrating a success, bickering, and bantering are all important activities in creating a design. And the designers will

include at various times a wide variety of people including the team members, outside vendors, and the customer. Design is more than a creative individual sitting at a terminal. Each participant in the process will have his view of what the design is, perhaps in the form of sketches, lists, analyses, prototypes, etc., but, during the design activity, the design is not in the possession of any one individual nor is there a unique description possible of it. It is in this stronger sense that he claims design is a social process.

Each participant in the design activity operates in what Bucciarelli terms to be his own "object world(s)." Within such worlds exist specifications and constraints which the person operating within it appreciates and has to communicate to the others. The participants in a design must therefore spend their time striving to create a design when none of them sees the artifact from the same point of view.

He describes finally three types of discourse that occur in the design process among the participants: *constraining, naming and decision discourse.* In the first different constraints are negotiated for the design, in the second a vocabulary is negotiated - for example, to name a part of the design - and in the third decisions impacting the design are negotiated.

Invariants of the Design Process [Goel and PIrolll,1989]

In a just published paper, Goel and Pirolli attempt to identify invariant structures in prototypical design problems from within information processing theory in cognitive science. They look at design from the point of view of an individual working alone to solve it. This work complements that of Bucciarelli. While others have suggested that design is the same as problem solving in general, these authors argue that design is a subset of those problems. It has particular characteristics that they call invariants which uniquely define the task environment and the problem space. The invariants they list of the task environment are:

- Size and complexity of the problem
- Goal statements as input, specification as output
- Temporal separation of specifications and delivery of product
- Delayed feedback on product performance
- Independent functioning of product
- Cost of action

- No right/wrong answers, rather good/bad answers
- Many degrees of freedom in problem statement

The invariants of the task environment, together with the limitations on how people can process information, cause the following invariants to exist in the problem space:

- Extensive problem structuring
- Extensive performance modeling
- Personalized and institutionalized stopping rules
- Limited commitment mode control strategy with nested evaluation cycles
- Making and propagation of commitments
- Solution decomposition into *leaky* modules
- Role of abstractions in transformation of goals to artifact specifications
- Use of artificial symbol systems

Both this paper and that by Bucciarelli argue that design involves looking at the artifact from many perspectives (whether by an individual or by a team), that different symbol systems are involved (these are used above to carry out the analyses indicated, for example) and that many false tracks will be followed. Goel and Pirelli point out that often these false tracks will be repeated later as they will have been forgotten. The discourse that Bicciarelli indicates will occur among the participants in fact also occurs for an individual when he has to translate from one view of the artifact to another, and, in fact, the individual is never working alone. He has to interface with the organization within which he is working. He will have to discuss the planning of his activities, the stopping rules, etc.

The Variety of Design Problem Types [Westerberg, 1988]

Finally we would like to make the point that the design process is extremely varied by the nature of the artifact (process or product) and by the context within which the process is being carried out. Thus many discussions about what the design process is are often confusing because the discussants are thinking of quite different "points" in the following space of design problem types.

We shall characterize the space first by the nature of the artifact to be designed and then by the context in which the design is to take place. Updating the ideas in Westerberg somewhat here, we suggest the following six characteristics are important: (1) whether the design is routine or creative (this one issue has often split whole conferences on design into two camps who refuse to listen to each other - in the first one strives for complete automation while in the second one generally considers the development of aids to support the designers), (2) the size of the design activity required (a door knob vs a Boeing 747 - if the activity will require teams of people to carry it out, then the organization and coordination of these teams is a major issue in the design), (3) the degree of coupling of the decisions made among the parts into which the design is decomposed (highly coupled decisions can, if the problem can be so formulated, lead to the effective use of such tools as large-scale mixed integer programming approaches), (4) whether geometry plays a significant role in the design (there is a whole research area on geometric representation as well as the designers' intuition that can be brought to bear If shape of the artifact is a significant aspect of its design), (5) how many are to be produced (one building, millions of memory chips - one cannot be sloppy about how to manufacture a memory chip when developing its design, it is too costly to recover from any errors), and (6) the degree that environmental issues play a role in the design (is one spending most of ones time trying to satisfy regulations).

The context within which the design is being carried out also impacts the approach to design. These include current costs for labor and materials, whether the artifact or parts of it can be purchased from elsewhere, whether the company has parts in stock that could be part of the design, the time available for carrying out the design, the skills and experience of the design team members, etc.

If one can place the artifact into this classification scheme, a conjecture is that the more suitable approaches for carrying out the design can be selected a priori.

In Summary of the Above

Design is impacted by all of the above: the characteristics of the players, the nature of the artifact and the context within which the design is to take place. A significant part of design is as a social process among many individuals. Any aids which are to support it must allow each player to have his own model of the activity; to have access to a variety of tools which can used in an

opportunistic fashion - i.e., when the player wishes to use them - and each of which may use a different symbol system; and, very importantly, the aids must improve the negotiation process.

Desired Characteristics for a Design Support Environment

After defining what we consider to be the preliminary design process as carried out by a team of designers and important issues related to it, we list a number of characteristics we believe a computer-based design support environment must have to support this activity.

(1) First it should be able to support persons using a variety of different programs, each of which may use its own representational scheme - i.e., it should support a variety of symbol systems - programs written in Pascal, in Lisp, in Smalltalk, etc. We would like it ultimately to have a common high level language within which these other programs are accommodated (see for example the description of the interface description language IDL [Snodgrass, 1989]).

(2) The system must accommodate every user having his own view of the object. Even if the system used but one language, every designer will have a view of the design that reflects his background, interests, and so forth, as discussed above.

(3) The system should aid in maintaining consistency among the different views.

(4) Every person using this system or creating one of the support programs will use his own naming schemes for creating these programs. There should be NO name collision problems. Name "negotiation" should be supported (Bucciarelli, 1988) as discussed above.

(5) Once an object has been created and placed in the system for others to see and/or use, it should be virtually impossible to change it or to remove it. Old objects not used anymore can be archived, but they should never be lost. This feature is to guarantee this part of the history of the process is captured. The project history is needed if one is to provide an explanation facility for why a decision was made and how it was implemented. Also if someone has used an

object and this object is later altered, he will almost certainly find his program has a bug in it that was not there earlier.

(6) In spite of the previous requirement that an object, once in the system, cannot then be altered, the system must support evolutionary design development and constant change without extensive rewriting; i.e., it must be possible to reuse easily extensive portions of previously created objects. We intend to argue that these two requirements can be accommodated rather nicely.

(7) A user of the system should be able to find things stored within it quickly and by a variety of methods. It will be a large system and things will easily be lost and/or misplaced in it, perhaps never to surface again if one is not careful. Also much of the system will have been created by others.

(8) A system such as this will be complex to browse without getting lost. The context of the information should be available easily to anyone attempting to look around the system.

(9) The objects within the system will be both active agents and passive data structures. Active agents receive inputs and compute outputs. Passive objects are information structures that represent different aspects or views of the design.

(10) It should be possible to create a fully automated portion of the design activity within the computer support system when the steps required are understood and the building blocks are available.

(11) It should be possible to have several users carrying out tasks simultaneously.

Capturing the Design Process - Design is Modeling

How can the design process be captured? We suggest that the activity of design is a form of continual model development by the participants, and therefore much of it can be recorded by providing a support system to capture and organize models as they are developed and shared.

Models are created at several levels of abstraction: information models in the form of memos which are subsequently structured, annotated and interrelated; models of the organization of the activity; models which are goal statements and functional requirements for the design; and models - either on paper or as prototypes - of the physical devices and systems built of them which might meet these requirements and various combinations of all of these models.

Impact of Computer Technology

The development of any future design aids, particularly for modeling, must also be aware of the technology available now and in the foreseeable future. Where the aids can exist on computers, there are advances in both software and hardware that impact how they might be structured. From the software point of view, we believe that significantly improved understanding of computer languages is taking place and that it is directly relevant. Relevant references in engineering design are: Piela [1989], Henning and Stephanopoulos [1989a, 1989b] and Sapossnek [1989]. We shall discuss this improved understanding and its impact on design aids in some detail. Clearly in hardware, many advances are expected. The impact of having a supercomputer on ones desk and of networks with superwide bandwidths which allow high resolution TV images to be transferred along with voice and data are going to have enormous impacts. We already see the impact of E-mail networks, the FAX machine and conference calls on the ability of groups of people to work together more conveniently that in the past.

We shall concentrate in this paper on advances in software, particularly in languages, as they are able to aid modeling in the very general sense that we have described above.

Languages for Design

A main thesis of this paper is that

> ***there are concepts in modern languages that will have a major impact on the form of future computer aides to support the design process.***

Such aids must support the easy reuseability and extensibility of previous code. It is important that we in chemical engineering become aware of them. In particular such concepts as "inheritance" and the notion of "prototypes" supported by the types of objects and operators that are appearing in object oriented programming and expert system shells are no accident. We shall look at our recent research work with the ASCEND modeling system to explain these concepts and then look at how they, along with some other concepts, can be used in the more general setting of creating a preliminary design environment.

The ASCEND System

We have been working for the past four years on the creation of the ASCEND (Advanced System for Computations in Engineering Resign) system [Piela, 1989]. The ASCEND system has three major components, a declarative modeling language, a solving system, and an interactive browsing system. It has been created specifically to permit complex algebraic (and, in the near future, mixed algebraic and ordinary and partial differential equation) models to be developed and solved by engineers much faster than is currently possible. A declarative modeling language in principle means that the modeler only has to state what is to be true at the solution to create a model; i.e., he only has to write the equations. From a practical point of view, he also has to provide some help to establish the initial guesses for the variables. This later step is inherently procedural. (While establishing initial guesses is extremely important, we shall not have time here to discuss this issue here. There are interesting mechanisms in ASCEND to support establishing initial guesses and more are planned in future versions.) Solving is a separate capability and is provided by the solving system in ASCEND. In chemical engineering jargon, ASCEND uses the equation solving paradigm for modeling.

As we are interested in the impact of languages, we shall only look at the modeling language for ASCEND in this paper.

General Characteristics of the ASCEND Modeling Language

The ASCEND modeling language is based on only a few concepts, and yet it seems to be very powerful for constructing complex models.

First it is a "strongly-typed" language; i.e., every object in ASCEND must have a type associated with it. Modeling in ASCEND is, in fact, the creation of type definitions. While requiring that every object has a type appears to place a burden on the modeler, in fact it allows many very subtle errors to be detected before model solving is attempted, such as incorrectly setting a temperature equal to a pressure or attaching a liquid stream to the vapor output of a flash unit.

ASCEND uses "inheritance" and a "part/whole paradigm" for building complex types. Starting with the fundamental types of reals, integers, booleans and units (e.g., m/s or lbs/ft^A3), the modeler creates complex type definitions by repeated "refinement" of previously defined types. The refinements will typically have new "parts" in them which are "instances" of previously defined types.

The simplest of user created types in ASCEND are "atoms." Atoms correspond essentially to types of variables. For example, "temperature," "pressure," and "molarJlowrate" are atoms that a modeler of a chemical flowsheet would be likely to define. Atoms have dimensionality (e.g., mass/time or dimensionless) and have parts which include a default value, a lower bound, an upper bound and several flags which indicate if the variable is fixed or calculated, has an initial value already guessed for it, etc. Mathematical relationships can be written among instances of atoms; for example one might write

```
flowl + flow2 * S*(G+100 {lbs/ft^A2/s});
```

as such a relationship. Here flowl and fk>w2 are "instances" of the type massjlowrate with dimensionality of mass/time; S is an area of dimensionality length^A2 and G is a mass flux with dimensionality of mass/length^A2/time. Note the distinction between dimensionality and "units." The constant 100 here has dimensionality of mass/length^A2/time and units of lbs/ft^A2/s. If this equation were dimensionally inconsistent, ASCEND would detect an error.

Complex types are called "models" in ASCEND. A chemical engineering flowsheet modeler would likely create a 1lash_unit," "pump," and a "stream" as models. A particular flash unit, FL1, is an example of an "instance" of the type flash_unit.

Inheritance is also used to check for modeling errors when the modeler might try to replace a part of his model with another part which is of the wrong type. (We will discuss this point later when we look at the operator IS_REFINED_JO.)

ASCEND uses flexible arrays. It permits the modeler to create arrays whose elements are instances of the same ("base") type but where each instance can be individually refined to be a more complex types; for example, the modeler can create a array of trays and then upgrade the seventh one to be a feed tray. Note that one is able to have a array of models instances, not just of variable instances as in Fortran or variable or record instances as in Pascal.

Finally, we built ASCEND on the principle of NO hiding of model definitions. For example, the modeler can use any variable which is inside a part of his model. Access is by constructing a "pathname" such as

FI1 .feed_stream.T

which says that we want the temperature of the feed stream in the flash unit FL1 in our flowsheet.

Operators in ASCEND

The four required and explicitly stated operators are illustrated with the following:

vapor__stream **REFINES** stream;
product! **IS_A** vapor_stream;
flash1.vapor__out, compressor!.feed **ARE_THE_SAME;** and
vapjn, vap_put **ARE_ALIKE.**

Two additional explicitly stated operators, which are there for convenience, are illustrated by the following:

UNIVERSAL MODEL methane_physical_properties; and
flashi .vapor_out **IS_REFINED_TO** hydrocarbon_vapor_stream.

Having only four required explicitly stated operators is one point of interest here. The "REFINES " operator allows one to set up model definitions through inheritance by using the definitions of already existing models. With the statement given above to illustrate this operator, the vapor_stream model will have all the attributes of the stream model plus those that make it a vapor stream.

The "IS_A" operator declares producti to be an instance of a vapor_stream. its use makes producti a *part* of the model in which the statement is being made.

The "ARE_THE_SAME" operator is the "merge" operator in ASCEND. By merge we mean that all items declared to be the same *will share the same storage locations and all redundant relationships will be removed* (only one copy of the equations will be produced). The merged items will have multiple names, any of which can be used to access them. In Borning's ThingLab project [Borning, 1979], the merge operator was stated to be both very powerful and very difficult to implement. Both observations are true. (This definition of merge should be clearly understood to be *very different* from an alternate definition which is the merging of lists, either by concatenation or logical intersection.)

The "ARE.ALIKE" operator used above indicates that whatever the streams vapjn and vap_out become in the future, they must both be of the same base type. For example, if someone decides that vapjn is to become a hydrocarbon stream, then vap_out will become one, too.

"UNIVERSAL" is included for convenience as its actions can be accomplished by repeated use of the ARE_THE_SAME operator. It means for this example that all instances which are of type methanejDhysical_properties will be merged. Finally, the "deferred binding" allowed by the "IS_REFINED_TO" operator can be accomplished by the following code:

```
he IS_A hydrocarbon_vapor_stream;
he, flashi .vapor_out ARE_THE_SAME;
```

In this example, we are making the vapor stream leaving flashi into a more refined type, hydrocartx>n_vapor_stream. Note, flashi is a part of our flowsheet, and we *havereached inside* it to make its vapor stream into a more refined type. The type hydrocarbon_vapor_stream must be a refinement of the type vapor_stream or ASCEND will report a modeling error.

There are two additional operators in the ASCEND which are there implicitly. These are "has_part" and "hasjrelationships." When one writes a model, one lists the parts and relationships within the model definition.

Examples of ASCEND Modeling

Figs. 1,2 and 3 illustrate models created in the ASCEND language.

Fig. 1 shows how easy it is to set up a column.section model with only a few statements. Note how we reach inside the parts of a model we are currently writing by using path names (e.g., stages[i+1].V_out). The writer of the column section model is using the tray model as a part and is able to access parts of the tray model. The *user* of the model defines the interface for it, not the *writer ot* the model.

Fig. 2 shows pictorially a portion of the structure that could arise in the creation of some stream and separator models. There is a basic stream model. It has three refinements: vapor, liquid and twojDhase stream models. The two_phase_stream model not only refines the stream model and therefore has all the attributes of a stream, it also contains two parts which themselves are streams: a vapor stream part and a liquid stream part. The flash model is a refinement of a simple_separator model. The simple_separator has three parts, each of which are of type stream: feed, prodi and prod2. In the flash model, the type of prodi IS_REFINED_TO a vapor stream and prod2 to a liquid stream. A part in the flash model is a vapor liquid equilibrium model (vie) which has two streams in it. These are merged using the ARE_THE_SAME operator with the two product streams in the flash unit.

Fig. 3 illustrates the power of flexible arrays. A model is written for a flowsheet at a single instant in time. One can next create an array of flowsheet models with the array indexed over a set of time steps. By including an array of Runge Kutta or modified Euler models to connect state variables in time step k to those in k+1 for all k, one has created a model to solve the time behavior of the flowsheet. The ability to solve ODE'S is written in ASCEND rather than by writing a new kind of solver. What one has taken advantage of is that ODE's are in fact solved by converting them into sets of approximating algebraic equations. If the flowsheet model is an initial value mixed algebraic/ODE system, the solver in ASCEND will discover that it can march from one time step to the next to solve (by partitioning and precedence ordering the equations).

The use of inheritance is a feature of many modern languages: C++, Smalltalk [Goldberg and Robson, 1983], Flavors and Commonloops and the expert systems "shells" like KEE [Filman, 1988], Knowledge Craft [Pepper and Kahn, 1986] and ART [Inference Corp., 1985]. Thus many languages now exist with the notion of object *refinement* as a means to develop programs. The merge operator is in Bornings THINGLAB, a language for constraint maintenance. It is an *absolutely essential operator* in ASCEND and is required for the deferred binding permitted by the IS_REFINED_TO operator. The merge operator could, of course, be implemented in any of the above languages and shells, albeit with some considerable effort. THINGLAB also emphasizes the use of the part/whole paradigm which is a main feature of ASCEND.

Unaware of the similarity to these languages at first, we developed ours for quantitative modeling using chemical engineering flowsheet development as a guide. We invented very similar operators to those that exist in many modern languages and shells. We believe that these operators will eventually evolve into a best set for the structuring of complex models.

A Proposed Design Support Environment

We now take these objects and operators and look at how they might apply to the creation of a design support environment for preliminary design. If they represent a first attempt at some sort of universal truth about modeling, and, if we can convince the reader that a design support environment is a modeling environment, then these operators and operands should appear to work very well in aiding this task. We start by suggesting how a complex model might be created within our environment.

Creating a Complex Model

Our scenario has a person on the design team working at a computer workstation. The screen will be his (or her) "canvas" on which the next model is to be created. He will first give the model a name so it can subsequently be filed. He will reach down into the computer environment to find whatever models (file structures, quantitative models, memos, active agents, etc.) are needed. These will be brought up onto the screen, each represented by an icon. Each will become a "part" of the model, equivalent to writing an IS_A statement in ASCEND. As each active

agent is brought up to the screen, it will be exercised to be certain it is well understood (in ASCEND each part of a model can be isolated and solved (by the solver, an active agent) to establish initial values for it, for example). Then the designer will "wire" this model into any parts already there to form a larger model and perhaps test it with an active agent. Fig. 4a illustrates a model at about this level of development. We can imagine the use of a mouse to click open the model into its own window so its details are exposed to find the parts to wire together. Once wired in, this larger system of parts may be exercised to expose any obvious wiring errors. One is programming but with models as the building blocks of the language. Once a model has been completed, debugged and found useful, it can be placed into the file system and made public.

We would like to contrast this approach to creating a model with the approach we proposed quite early in this work and which we have rejected as not supporting our current vision of the design process. In our original approach (see Fig. 4b), we suggested a model would be created in an orderly fashion from the top down or bottom up. First, there would be a set of memos written, annotated and organized. Next we suggested the writing of a set of specifications, with the memos being included as a part of these models to supply the background documentation. Then these specifications would become part of the goal models for the design, which themselves would become a part of the models that represent the actual physical components from which the design is constructed. At each level of abstraction, the higher level is explicitly a part of it. Note that here the most abstract models (the memos explaining purpose of the design and background for it) are being created first - from the top down - but the models to capture them can be writtin in a bottom up style.

While this approach to modeling is appealing, it is not what we are advocating. Rather we are suggesting that the designer will create separately, and often in a somewhat random order, many alternative models of the specifications, of the goals, of the background documentation and of possible physical systems which might meet the specifications. Some of them will reference earlier models as a part. But the order of concept generation is not just top down. Much of our modeling is by gathering together different candidates from these building blocks and wiring them together *from outside them* - they become parts which are then wired together to form a particular design alternative. We would like an abstract separator in a goal model to become an extraction process in one version of the model and a distillation based process in another - without altering the code of the goal model that has the abstract separator in it. The higher levels of abstraction for the design can be thought of as providing constraints on the design while the physical models can be thought of as providing the capabilities. Together one has the relationships that define

the design. Both the goals and the physical models are part of the picture painted on our design "canvas" - rather than the physical models owning the goals as a part of them.

Proposed Approach

To design an environment with the attributes described earlier and which permits the above scenario, we propose the following ideas *which are at best only a start at exposing the concepts needed.* As indicated earlier in this paper, we are stepping out onto "thin ice" as the ideas presented here have not yet been implemented and tested. However, we offer them to illustrate with more than just handwaving some of the issues that need to be resolved to create such an environment.

What is a Model?

A model will be a collection of parts, some of which are active agents and some of which are passive information structures. The definition is recursive; active agents themselves can be models.

Path Names

Each user or subsystem developer within the environment will create his (or her) models or programs using a set of names he wishes to use. A means to avoid name collision is to require that every name includes the path to it. Examples of path names are in Fig. 1 as they are used in the part/whole modeling in ASCEND. For example, the name stage[i+1].V_out is a path name. We created an array of stages to be a part of the column_sectk>n model, each being a tray model. V_out is a part of the tray model and is itself a vapor_stream model. While the name V_out belongs to all trays in the system, the particular one on the i+1st stage is uniquely identified by its path name: stage[i+1].V_out. If our column section became a part called "top_section" of a column model and the column model part of a flowsheet called "benzene_column," then the temperature of the vapor stream of the 5th stage in the top of the column would be called uniquely:

benzene_column.top_section.stage[5].V_out.T

It is by the use of path names in ASCEND that all names are unique. Also the use of path names significantly reduces the need for the invention of names. It is then by use of the merge operator, ARE_THE_SAME, that the modeler can "wire" parts together, but the wiring is an *explicit* act on the part of the modeler. For example this temperature and T_contro! can be made the same by stating:

T_control, benzene_cx>lumn.top_section.stage[5].V_out.T ARE_THE_SAME;

In many other approaches to naming, the names themselves are used to accomplish the wiring. For example, the name "temp* might be a negotiated name for temperatures by all modelers. The designers have to negotiate the names before modeling if they adopt this approach. The system identifies temperatures by finding this name and could at times even equate them automatically if some agreed set of qualifiers match. We suggest this *implicit* typing and wiring is restrictive and actually quite dangerous. As Bucciarelli stated in his article, designers spend a significant amount of time negotiating names. To support this negotiation we suggest that each use his own naming scheme; with path names, each name is guaranteed to be unique. Then the result of the negotiation is to wire them together explicitly with a merge.

We can extend the idea of path name to include the file system within which we store our models. The flowsheet model might be stored within a hierarchical file structure (like a UNIX™ file structure). A unique name for this temperature might then be:

//usr/aw0a/ascend/models/flowsheet2.
benzene_column.top_section.stage[5].V_out.T

where the // indicates the root node of the file system. The T delimiter between subdirectory names indicates paths through the file system while the V indicate delimiter indicates paths through parts of the part/Whole hierarchy into the (ASCEND) model.

We are therefore suggesting that our environment should provide access to any variable anywhere by its path name, including the path through the file system and then into its parts.

The File Structure Itself is a Model

We would like to remind the reader that the organization of the file structure is itself a model. Each user evolves her (or his) way to organize the information that she is creating or using. The files are stored in a liar file, with their location on disk memory kept in a directory. Different users place each of the files into any directory they choose by having the system simply keep the directory /subdirectory tree as a set of links (pointers) to the files placed in it. This tree is then kept as a model belonging to the user. Note that *the same file can belong to several file structures;* it will have a number of different names resulting, one for each path to the file.

If the user has made her file system public to others on the design team, others will begin to access models within it and to wire them into their models. When (note, we do not say "if") the owner of the file structure decides to reorganize it, models using the existing file structure and its path names will become obsolete. However, if the file structure is treated as a model, it can be kept. The user reorganizes simply by creating another model which represents her new file structure. Past models are not destroyed.

Finding Things

The modeling system we shall create will be very large, and there will be a significant problem in finding things within it. We will need to locate previous models and parts within previous models both within our own files and in those of others, and we will want to do this in a variety of ways. Finding things can be by name, by attribute name and/or value (i.e., associative searches as in production systems like OPS5 or in database systems) and, not to be overlooked nor underestimated in importance, by *browsing.* Any mixture of these could be used in any one search.

Many modern programming systems support browsing as a means to aid a programmer. The key way one programs using LOTUS123 or EXCEL illustrates an effective use of browsing. One remembers approximately where something is kept in the spreadsheet, looks around for it, and then places the cursor over the item. With a keystroke the item is inserted into the program statement being constructed. No name was used to encode it into the statement. A problem with browsing is that one can easily become lost because one cannot see the context within which one is carrying out the search. One means to overcome getting lost is to provide, in a separate

window on the screen, a picture of where one is, e.g., such as Fig. 2. Ones location in the main window could be shown by highlighting the appropriate icon in this window. One might also provide a means to see all path names to an object that one is in as a means to locate oneself, a mechanism used in ASCEND.

Control

We have not spoken yet about the control of the execution of the model being constructed. The modeler can of course decide at every step which agent should execute next. If there is to be any part of the system which is to proceed *automatically,* we suggest that this issue is quite complex and that ideas coming from the execution of logic-based systems, expert systems (see, for example, Tong [1987]), standard computer program control (with IF, WHILE, FOR and DO statements) all can play a role.

To illustrate how the control in expert systems might play a role, we briefly consider an expert system we are working on to synthesize separation processes. We want execution of this program to be very opportunistic in its behavior. At each step we would like it to "know" where it would be best to try its next computation. The control model is that described in Lien [1988a,1988b] for the system called AKORN-D. Briefly, it is a blackboard model in which all active agents bid for attention. Some of the active agents affect who will win the next bid and are called control agents; others work on solving the problem. When an agent wins its bid for attention, it executes. Then all a bidding cycle starts again. At issue is how such a system recovers from dead ends, etc. Clearly these issues can get very complex, and we will not discuss them further in this paper.

For the present we can simply assume the designer creating the model will manually control the execution of his model.

In this respect the work by Talukdar and coworkers [Papanikilopoulos and Talukdar, 1989] in a project called FORS is relevant. This work is to set up an environment in which both "aspects* and "operators" are given equal status as objects. Aspects are different views in the form of data about the artifact being designed; operators are agents that transform one aspect into another. In the FORS systems, the designer can be an agent in the system, too. At any time the system can suggest to the designer which agents can be invoked or which aspects are complete enough to

allow an operator to run, considering the current state of the design. If there are several aspects or operators which can be considered next, the user picks one. There is an icon-based used interface so the user picks operators and aspects by clicking on them using a mouse. The path through the set of operators and aspects is remembered and can be repeated after some of the variable values are changed in any of the aspects. The operators and aspects can reside on any one of several UNIX™ based machines; thus the interface may be on a Sun 3/60 while a finite element analysis routine corresponding to one of the operators executes on an HP Bobcat and the synthesis program that generates a "stick" model on a microvax II workstation. They can be written in any of several languages: OPS83, Fortran, LISP or C. At present this system is directed more to routine design than creative design.

Nothing Made Public Can Be Deleted

Members of the design team will need to share each others models. There will have to be a formal mechanism to place a model into the public domain for others to see and use. One designer may produce a model that another finds useful and she will then make it a "part" of her model. Access to the model can be by path name as we just discussed. To avoid modeling catastrophes then, the models placed in the public domain must remain there. They should not be altered. We now discuss how this feature can be allowed without destroying the need to continually evolve the models in the system.

Deferred binding from the ASCEND system provides us with an approach.

Deferred Binding

In ASCEND models, the user of a model can reach down into the insides of the parts from which a model is constructed by using a path name. There are two types of things inside: *other parts* and the *types* of the those other parts. For example the statement:

```
m1 IS_A boiler_memo;
```

in Fig. 5 declares the existence of the part, which is called m1, and the type of the part, which is "boiler_memo." Suppose the creator of the boiler_memo decides to add some labeling to it. If

that alteration is simply to add embellishments to it, then the alterations can be done by refining the memo using the REFINES operator. The original memo may, however, have become part of the documentation for model.a which has already been put into the public domain. ModeLa therefore cannot be altered.

Suppose the person who wrote modeLa becomes aware of the embellishment to the original memo, looks at it, likes it and wants to upgrade modeLa. A revised version of model.a is created that is REFINES the original one. The coding is

```
MODEL model_a_rev2 REFINES modeLa;
m3 IS_REFINED_TO labeled_memo;
END newjnodeLa;
```

and with it she has created model_a_rev2 that is a version of model a that includes the labeled_memo.

E-mail Can Tell People

Whenever a model is refined or a refined version of it is further refined, all users of that model can be automatically notified by E-mail. The refines link would have to be a two-way link so it could be followed to all users. This notification would occur when the refinement is placed into the public domain and could easily be done automatically. Of course the E-mail generator should have mechanisms in it so only those wanting to be notified are.

"Wiring" In Agents

Models will consist of both active agents (procedural code) and passive (declarative) data and information structures. Many of the agents, as we stated above, will use different symbol systems. We noted earlier that we would like to have, at the higher levels in our system, the ability to tie these agents and data structures together. We would like to use a common language based on the operators we have been discussing in the ASCEND system. The resulting models would be what we can call "task networks[1]* (not unlike the FORS approach [Talukdar, 1989]).

To allow these agents to become a part of our system, an interface has to be created around them that makes their internal variable structures accessible to the outside world. Then the path naming mechanisms discussed above can be used to tie the agents and data structures together using the merge operator to create larger models. Deferred binding can also be used to modify a network. For active agents, the variables will need to be partitioned into those which are input variables to the agent and those which are output variables. A task network would be a *model tor* carrying out part of the computations needed to perform an analysis of the design, for example. It could be repeated later with modified input. An *instance* of such a network would be as a "placeholder for the computations whose parts would become instantiated as the active agents are executed.

Explaining Things

Lastly, explaining things is done in large measure by documentation. We have suggested here that memos, which can be annotated, labeled, refined and wired together rather like in a hypertext structure [Garg and Scacchi; 1987, 1988] are a significant part of our design environment. These can become a part to any model. Wiring them in with an "ARE_RELATED_TO" operator (this operator only ties two things together - it would be used when the types of the operands are not relevant) allows anyone looking through a model to find its supporting documentation. We see an interesting "indexing" taking place. The memos, by being wired into the more quantitative models, are an index into the quantitative models. The wiring also make the quantitative models an index into the memos. They each can be used to aid in browsing the other.

A Summary of the Issues

We close with a rough list of the major "modeling" issues for our support system: *representing things, finding things, controlling things, recording things and explaining things.*

References

Borning, A. (1979), THINGLAB - A Constraint-Oriented Simulation Laboratory, PhD Thesis, Dept. of Computer Science, Stanford Univ., Stanford, CA.

Bucciarelli, L.L. (1988), "An Ethnographic Perspective on Engineering Design," in Schon, D.A. (ed), Design Studies, Vol. 9, No. 3, pp 159-168, July

Daniell, J. and S. W. Director (1988), "An Object Oriented Approach to CAD Tool Control within a Design Framework," ECE Dept.

Filman, R.E. (1988), "Reasoning with Worlds and Truth Maintenance in a Knowledge-based Programming Environment, Comm. ACM, Vol. 31, No. 4, pp382-401.

Fisher, A.S. (1988), CASE: Using Software Development Tools. John Wiley & Sons, New York.

Garg, P.K. and W. Scacchi (1987), "On Designing Intelligent Software Hypertext Systems," CS Dept. USC, Los Angeles, CA, submitted.

Garg, P.K. and W. Scacchi (1988), "A Hypertext System to Manage Software Life Cycle Document," CS Dept., USC, Los Angeles, CA .

Goel, V. and P. Pirolli (1989), "Motivating the Notion of Generic Design with Information-Processing Theory: The Design Problem Space," AI Magazine, pp18-36, Spring.

Goldberg, A., and D.G. Robson (1983), Smalltalk-80: The Language and Its Implementation, Addison-Wesley.

Henning, G., H. Leone and G. Stephanopoulos (1989a), "MODEL.LA. A Modeling Language for Process Engineering Part I: The Formal Framework," Laboratory for Intelligent Systems in Process Engineering, Dept. of Chem. Engng., MIT, Cambridge, MA.

Henning, G., H. Leone and G. Stephanopoulos (1989b), "MODEL.LA. A Modeling Language for Process Engineering Part II: Multifaceted Modeling of Processing Systems," Laboratory for Intelligent Systems in Process Engineering, Dept. of Chem. Engng., MIT, Cambridge, MA.

Inference Corporation (1985), ART Users Manual, 5300 W. Century Blvd., Los Angeles, CA 90045.

Lien, K.M.(1988a) (1988a), Expert Systems Technology in Synthesis of Distillation Sequences, PhD Thesis, Univ. of Trondheim, Trondheim, Norway.

Lien, K.M. (1988b), "Expert Systems in Chemical Engineering - On the Organization of Problem Solving Computer Programs," Chemdata88, Gothenburg, Sweden, June 13-15.

Papanikolopoulos, N., and S.N. Talukdar (1989), "FORS: Flexible Organizations," EDRC Report , Carnegie Mellon Univ., Pittsburgh, PA .

Piela, P. C. (1989), ASCEND - An Object Oriented Environment for the Development of Quantitative Models, PhD Thesis, Carnegie Mellon Univ., Pittsburgh, PA 15213

Penzias, A. (1989), Ideas and Information, Norton&Co., New York.

Pepper, J. and G. Kahn (1986), "Knowledge Craft: An Environment for Rapid Prototyping of Expert Systems," Proc.SME Conf. on AI for the Automotive Industry, Soc. of Manuf. Engineers.

Sapossnek, M. (1989), "Research on Constraint-based Design Systems," Proc. 4th Internl Conf. on Application of AI in Engineering, July.

Snodgrass, R. (1989), The Interface Description Language Definition and Use, WH, New York.

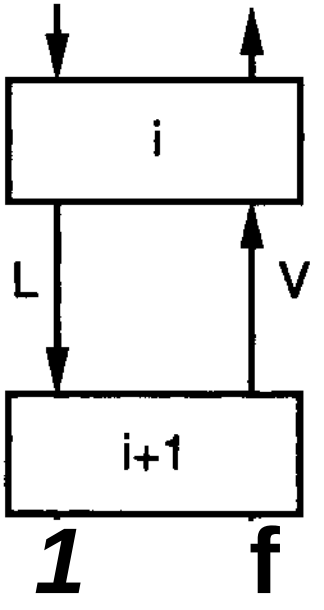
Stephanopoulos, G., J. Johnston, T. Kriticos, R.Lakshmanan, M. Mavrovouniotis and G.Gilette (1988), "DESIGN-KIT: An Object-oriented Environment for Process Engineering," Comp. chem. Engng, Vol. 11, No. 6, pp655-674.

Subrahmanian, E., G. Podnar, W. Elm, and A.W. Westerberg (1989), Towards a Shared Computational Support Environment for Engineering Design," EDRC Report, Carnegie Mellon Univ., Pittsburgh, PA.

Tong, C. (1987), Toward and Engineering Science of Knowledge-based Design," Internl J AI for Engineering, Vol. 2, No. 3, pp July.

Westerberg, A.W. (1988), "Synthesis in Engineering Design," Chemdata88 Conf., Gothenburg, Sweden, June 13-15.

```
MODEL tray;
    • • • •
End tray;


MODEL column_section;

    ns                          IS_A positivejnteger;
    stage[integer]              IS_A tray;

    FOR i:1..ns-1 CREATE
        stage[i+1].V_out,
        stage[i].V_in           ARE_THE_SAME;

        stage[i].L_out,
        stage[i+1].L_in         ARE_THE_SAME;
    END;

END column_section;
```



Column Example

Fig. 1 ASCEND Model for a Column Section

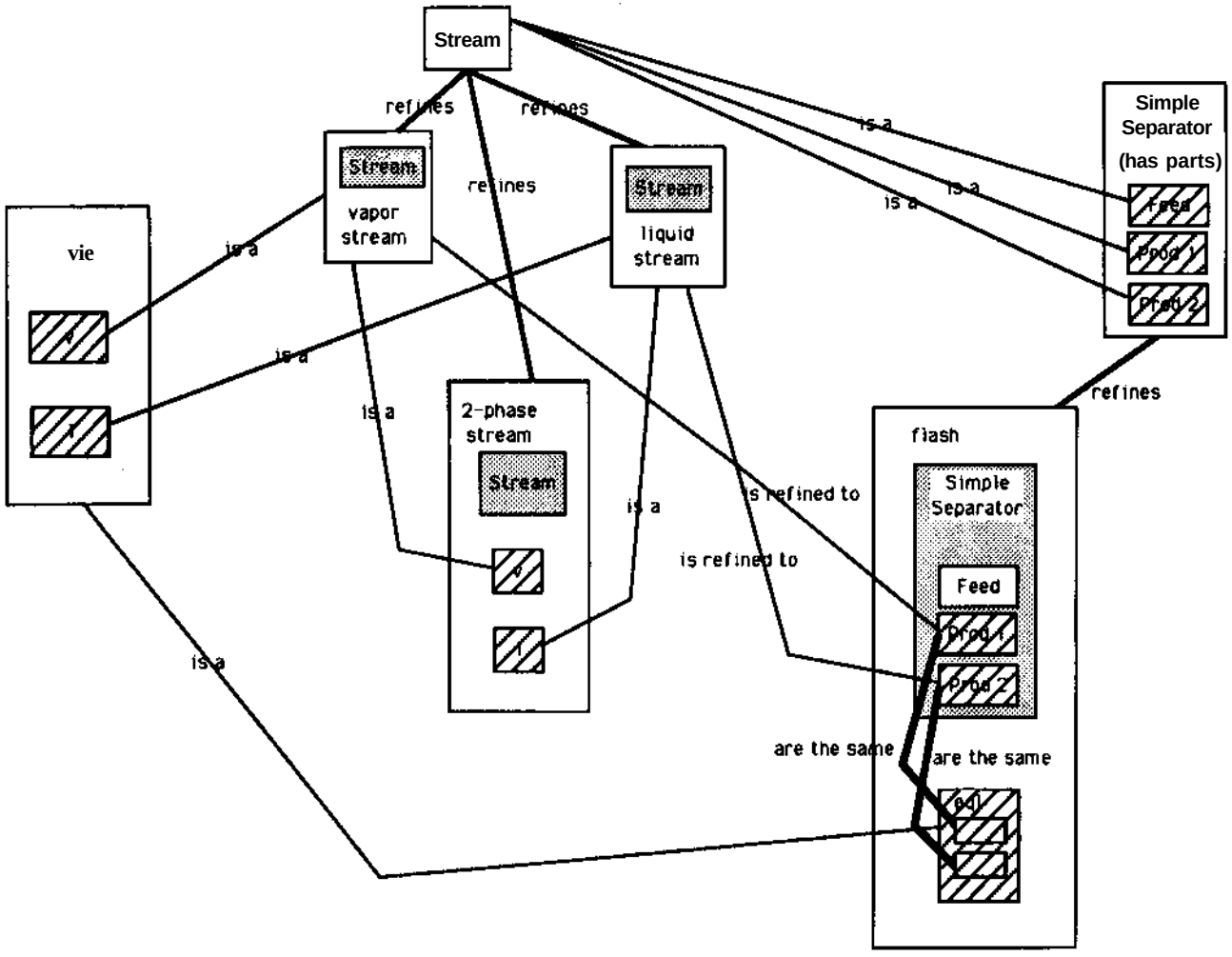


Fig. 2 Pictorial Representation of an ASCEND Model

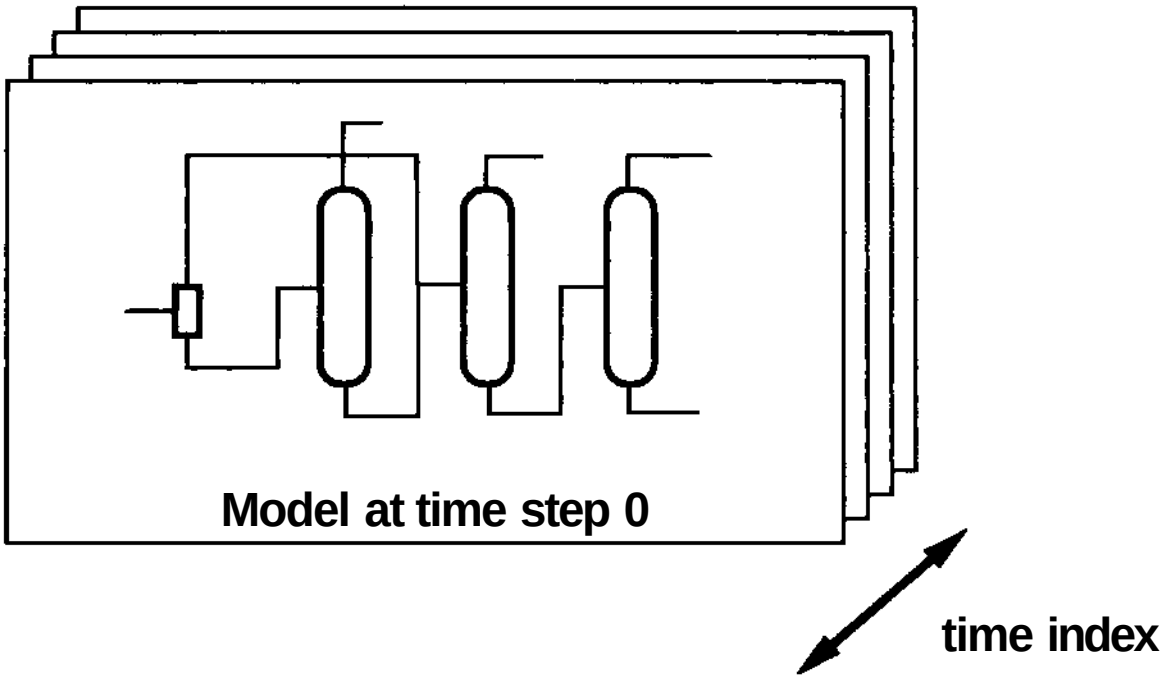
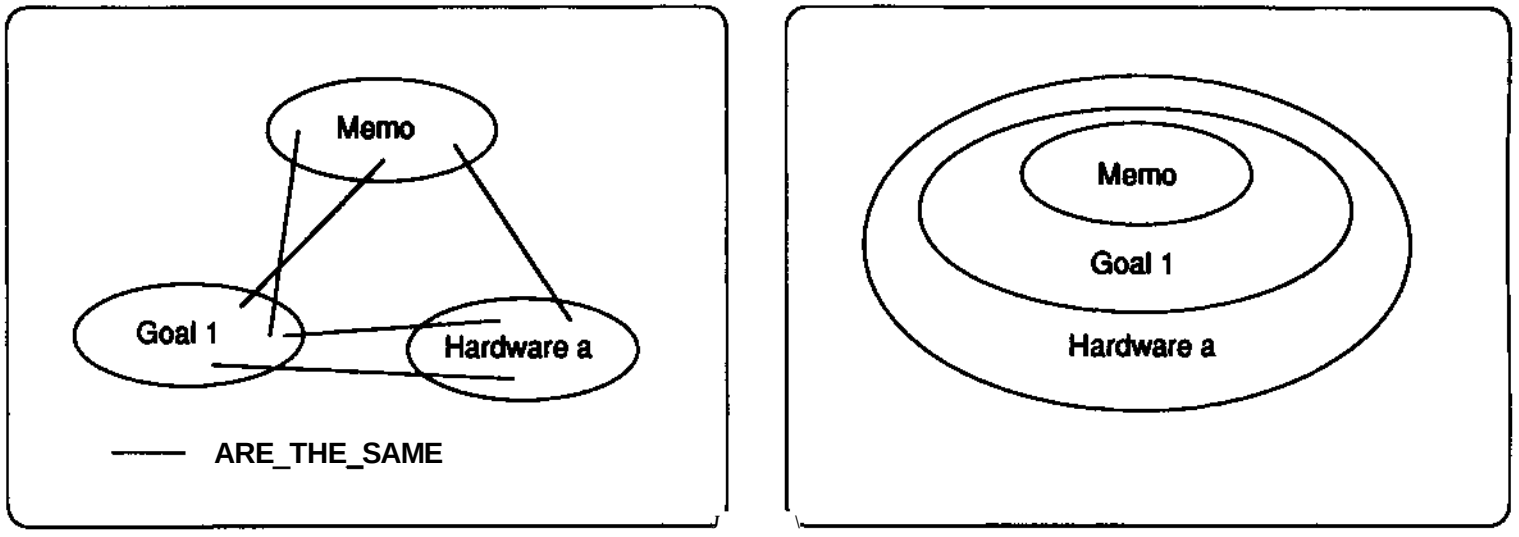
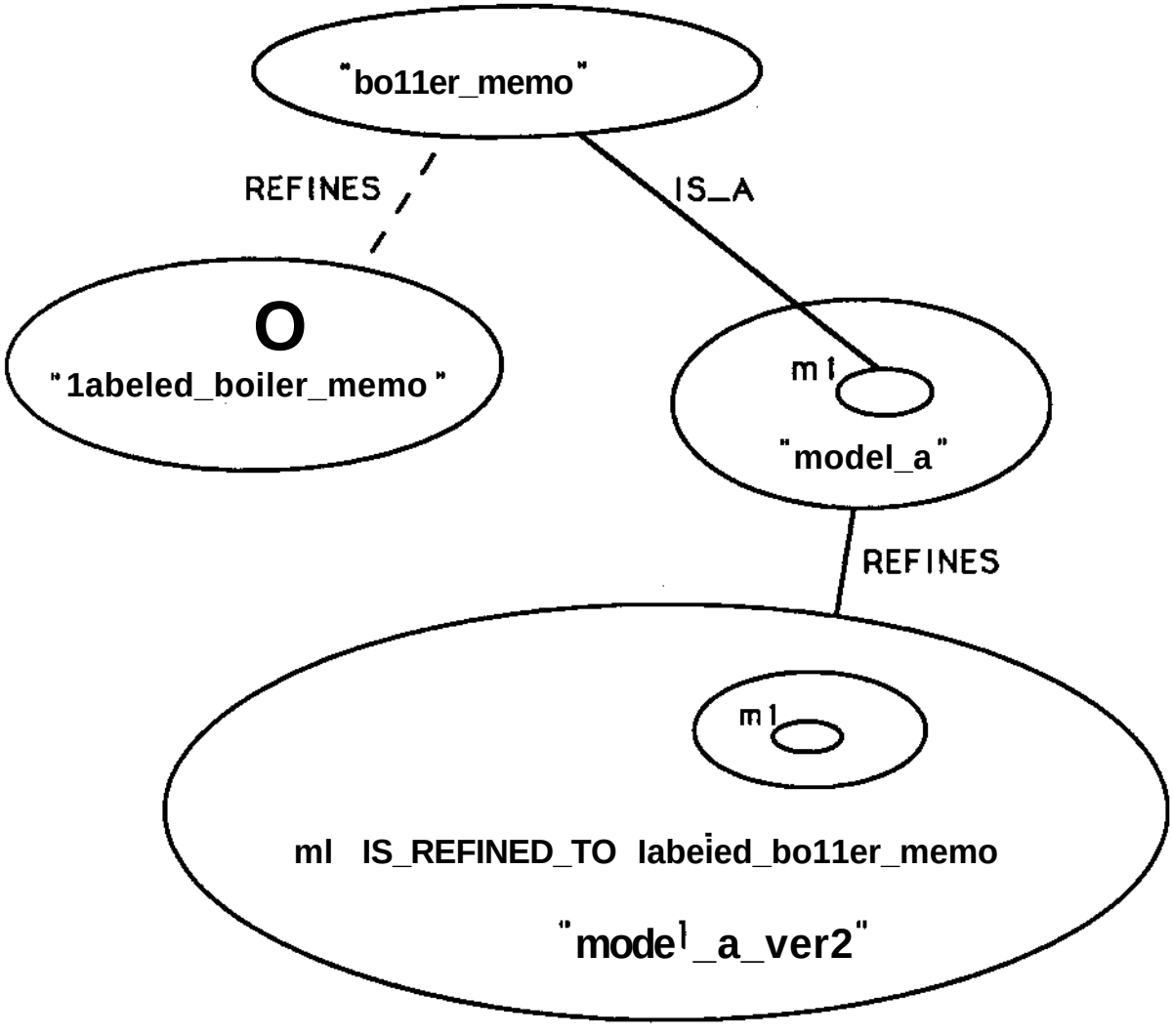


Fig. 3 Mixed ODE/algebraic Model in ASCEND



(a) (b)

Fig. 4 Modeling for Design, (a) As proposed here, (b) Top down only approach - not as proposed here



REFINES for labeling
REFINES and IS_REFINED_TO for uoaradina

Fig. 5 Incremental Modeling While Not Altering Existing Models