CARNEGIE-MELLON UNIVERSITY

# VERY LARGE SCALE MODELING

**A DISSERTATION SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS**

**for the degree**

**DOCTOR OF PHILOSOPHY**

in

**CHEMICAL ENGINEERING**

**by**

**Kirk André Abbott**

**Pittsburgh, Pennsylvania**

**March 1996**

*To Mom and Dad*

## Abstract

Very Large Scale Modeling (VLSM) requires a harmonious blend of language, algorithms and tools to be done successfully and efficiently. The sheer size of the modeling problem raises issues which are trivial when dealing with small problems. This work addresses the problems associated with formulating, representing and solving problems with up to 250,000 nonlinear differential-algebraic equations on a workstation class machine. The primary hypothesis is that for this undertaking to be successful, a dedicated VLSM environment is required.

A requirements analysis of a VLSM environment is performed and the primary issues of memory, speed and complexity are recognized. A review of the state of the art in modeling environments shows that they are deficient in satisfying the needs of a VLSM environment. A strongly typed, object-oriented, equation based language, ASCEND IV, which is based on the ASCEND III language, is designed to fulfill this role. Using partial implementations, ASCEND IV has been tested on modeling problems with up to 86,000 variables and has demonstrated its potential as a language capable of supporting a VLSM environment.

The large sparse linear algebra subproblem which is found in the solution of large systems of equations, can dominate the cost of solving such systems. The current approaches for reducing the linear algebra cost are reviewed. The deficiencies of these approaches are highlighted. New partitioning and reordering algorithms that exploit the hierarchical structures inherent in a structured modeling languages such as ASCEND IV, are presented. These algorithms are shown to give faster reorderings, much lower fill, reduced operation count and faster matrix factorizations. In a detailed comparison, codes which can exploit these better reorderings are shown to be competitive with state of the art linear algebra codes.

# Acknowledgments

I would like to thank Dr. David Chinloy and Mr. Richard Holzwarth of Alcan International for providing my first introduction to modeling. My appreciation goes to Professor Myung Jhon for giving me an opportunity to pursue graduate studies at CMU.

Many thanks to my thesis advisor Professor Arthur Westerberg, with whom I have had many interesting (at time fiery) discussions. He has always managed to goad me into achieving more. To members of the ASCEND group, Bob Huss, Joe Zaher, Mark Thomas and Boyd Safrit for their patience with me and my code. I would like to thank Tom Epperly for providing me with an excellent platform, the ASCEND compiler, for starting my work and his continued assistance over the years. My appreciation goes to the n-dim group for making tika.ndim available for my use. I am greatly indebted to Professor Mark Stadtherr and J. Mallya of the University of Illinois for the many discussions on large scale linear algebra and for providing the **LU1SOL** code.

To my friends Jacqueline, Lise, Chris, Juan and Jan, and Paula for helping to maintain my sanity. To my brothers Rohan and Anthony and my parents, for all their support over the years. To Debbie, whom has never lost faith nor love despite the distance.

I would especially like to thank Ben Allan, Sean Levy and Arletta for the roles that they have played in my life. To old Ben for his friendship, patience, knowledge and ever cynical attitude towards life. To Arletta who has treated me like family and showed me a different way of living. To Sean for being a brother, philosopher, great hacker and friend.

To all, I again say my thanks.

And for those who made it more difficult, thanks for helping to build character.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

## 1.1 Design

The design of a large-scale system is a complex and difficult process. Design is an optimization problem. Some measure of *goodness* is sought from the artifact being designed, subject to the constraints of time, man power, computational resources, finances and knowledge. The difficulties arise from the size of the problem and the millions of pieces of information that need to be collected, manipulated and analyzed over the course of a design. Invariably the process will involve many designers and may take years to complete.

Modeling and design are closely related. Modeling is the process of mapping reality into a representation that is thought to be useful for understanding that reality. Designers use models to aid in their decision making. A designer/modeler is then one who practices the art/science of modeling. Though there are many different models that may used over the life time of a design, computational or mathematical models are among the most important of these models and are the primary focus of this work. In mathematical modeling, the problem to be modeled may be represented as set of variables and the mathematical relationships among these variables.

Very Large Scale (VLS) systems are very difficult to design and operate monolithically. A concise and accurate definition of what constitutes a very large scale system is also difficult. Qualitatively a very large scale system is one that is too large to be solved efficiently as a whole. The process of attempting to understand a VLS gives rise to very large scale modeling problems. Large scale systems abound; the engineering disciplines provide a rich source of large scale systems and problems. In electrical engineering, the design of complex electronic circuits has given rise to a new acronym, VLSI (very large scale integrated) circuits. Chemical engineering design and operation similarly has its share of large scale systems. The example that most readily comes to mind is the design and operation of a petrochemical complex. Likewise a detailed dynamic, thermal, flow and vibration analysis of a single heat exchanger in such a design is just as formidable a problem.

The difficulties associated with the design of large scale systems has given rise to *design research*. Arising from this research are different design theories, methodologies and software tools for making the design of large-scale systems more tractable. A detailed discussion of design research is beyond the scope of this work. However, all the methodologies involve decomposition of the design problem and the subsequent coordination to achieve the desired goals of the design. The decomposition may be functional, hierarchical, spatial or by domain.

Ponton [Pon94] in arguing for a design *environment* for process engineering points out that the activities of

- *decomposition and aggregation*
- exploration
- evolution
- cooperation
- interaction

- automation *and computation*[1]

need to be supported [BA94]. Environments such as epee [CFSB94] and *n*-dim [LSKC93] have developed to support these activities which may be considered *design in the large*.

### 1.1.1 VERY LARGE-SCALE MODELING

Other design research assumes that a decomposition of the overall design problem has been done and concentrates on the modeling and solution of the computational models associated with subproblems of the design. In the design of a large petrochemical complex for example, a domain decomposition is normally performed such that process design is first performed and is later followed by mechanical, civil and electrical engineering design. However, even within these subproblems a very large computational model still needs to be formulated and solved and will involve multiple designers. If taken as a whole the sheer size simply overwhelms the modelers and computational resources.

Lasdon's, *Optimization Theory for Large Systems*, [Las70] published in 1970 discusses techniques which are still in use today for dealing with large-scale systems. The seminal work of Mesavoric, published in the same year [MMT70] develops theories to formalize *hierarchical decomposition*, a technique which has been used formally and informally for years to handle the complexity of large systems. Haimes [HL88], points out that the fundamental characteristics of large-scale systems is their multifarious nature; the structure of such systems (and they *all* have structure) may be multilevel, multistage, hierarchical and dynamical.

What constitutes a large-scale model varies with the class of problem being modeled. At the time of writing linear programs with millions of constraints and special structure are being solved [Wri93]. Global optimization problems on the other hand seem to be restricted to below a 100 variables though problems with 2500 variables with special forms have been solved [IP95]. For differential

1. The emphasized items are the authors embellishments.

algebraic systems there is at least one published benchmark of approximately 76,000 variables [ZBLZ95] for a full dynamic model of a chemical engineering process done using the SpeedUp [spe90] system on a Cray C-90 computer. Kelly et al [KFD91] have reported the solution of nonlinear optimization problems involving 35,000 variables. This problem was the real-time optimization of an ethylene plant. Recently the author has heard of a similar problem with over 400,000 variables being solved (private communication with D. Varvarezos 1995). The details of the modeling and solution of both of these examples were not given.

The literature reports very few hard numbers on the resources required to model realistic large-scale problems. The benchmarks published describe solution statistics such as problem size, number of nonzeros and the number of iterations taken for solution. Very few statistics are available on resources (personnel and computational) to perform the activities of

- developing a model formulation.
- encoding and realizing the model.
- debugging, scaling and initializing the model.
- analysis of the solution.

Frtizson et al, [FF92] in appealing for high level programming support in mechanical analysis, do provide some data. They describe the design of code to solve a saw-chain optimization problem. Using mainly low level tools (FORTRAN code) it took 2 man years of modeling effort. Another 2 man years was spent in exploratory design. The FORTRAN code had to modified over 1000 times. The solution time was 5 hours on a Sun 3/60. The problem had 73 equations.

The modeling activities along with the solution of the model are repeated over the life time of a design project. It may be argued that the modeling activity will consume more resources that the solution time over the project horizon.

In the absence of firm data it is necessary to use one’s own experience and *tangibles* such as the raised eyebrows of workers in the field of modeling to establish a basecase. The state of the art in modeling and solving arbitrary problems in chemical engineering seems to be generously bounded from above to about 20,000 variables.

### 1.1.2 TOOLS AND ENVIRONMENTS

Many disciplines, including chemical engineering, have attempted to use computer *tools* and to a lesser extent *environments* to aid the design process. Ponton [Pon94] defines a tool as a *software package designed to carry out a well defined although often complex task or set of related tasks* and an environment as a *software system designed to support the use of one or more tools*. Tools used in design are process simulators (AspenPlus [Asp93]), matrix generators (**GAMS** [BKM88]), linear algebra solvers (ma28 [Duf77]), differential equation solvers (LSODE [RH93]) and language compilers (FORTRAN and C). Environments include computer operating systems (DOS and Unix), process simulation packages (AspenPlus [Asp93]) and modeling systems (ASCEND [PMW93]). The distinction between tools and environments is often not clear; many tools have evolved into environments with the attendant advantages and disadvantages of evolving systems.

Some tools particularly in the *computational* aspects of design have been tremendously successful. These include the process simulators, most of which were based on the *sequential modular* solution technique. However very few tools or environments exist for supporting other critical areas of design, such as exploration and evolution.

Mattsson and Andersson [MA91] note that mathematical modeling is a time consuming affair but can be greatly facilitated by the use of proper tools. They strongly support an *object-oriented* framework for modeling through the Omola project. Geoffrion [Geo92] envisions a new generation of modeling environments that try to meet the design challenges of 1) a conceptual framework for thinking about models, 2) an executive language based on this framework for representing models and software integration on a large scale.

Piela in his Ph.D. thesis [Pie89] discusses at length the deficiencies of the state of the art in 1989, in modeling tools. The result of his work was the development of the ASCEND system which is based upon an objected-oriented language, where he reports an order of magnitude reduction in the time taken to formulate and debug models, when compared to traditional approaches. The ASCEND system attempts to address the issues of exploratory and evolutionary design. These design activities require flexibility in the types of problems that may be solved and flexibility in the specification of what variables may be solved for. The sequential modular strategy does not offer such flexibility. In the design of the ASCEND system, a conscious decision was made to use an equation-based solution strategy which largely decouples the posing of a problem and its solution.

A detailed discussion of sequential modular versus equation-based modeling techniques may be found in Westerberg et al. [WHMW79]. It suffices to mention that equation-based solution techniques, apart from facilitating exploratory design, introduce the potential for increased solution efficiency over the sequential modular strategies. Pantelides and Britt [PB95] discuss multipurpose process modeling tools and environments based on equation-based technology. Recognition of the potential benefits has led to the development of equation-based modeling tools such gPROMS [Bar92], **GAMS** [BKM88], AMPL [Gay91]. Some novel applications for these tools have been developed. Zenios, [Zen90], in trying to aid the formulation and solution of large-scale network optimization problems, remarks that a high-level modeling language such as **GAMS** can be used as a prototyping environment for algorithm development. This approach has been used by Paules and Floudas [PF89] for implementing algorithms for Heat Exchanger Network (HEN) synthesis.

All of the above tools use different proprietary *modeling language*s in which a model may be posed. This has prompted some workers to call for a universal modeling language standard [MA91].

Equation-based modeling and solution techniques have introduced new problems. In particular, at present, they require significantly larger amounts of memory than the sequential modular strategies and require that the modeler manipulate significantly more information. Object-oriented systems have emerged to manage the information swell, but further increase the demands on memory.

Arguably the state of the art in modeling software at the time of writing is an object-oriented, equation-based modeling environment such as ASCEND or Omola.

### 1.1.3 THE FUTURE

In the last few years, a number of changes have taken place in the world in which we live which will irrevocably affect the way that modeling is conducted. These include

- a drastic and continuing reduction in the cost of high speed memory in computers, and at the same time an increase in the speed and number of processors.
- the explosion in information technology, via systems such as the World Wide Web.
- increased awareness of the need to protect the environment, and the dwindling of naturally exploitable resources.
- increased competition in the global marketplace.

At first it may seem that some of these are unrelated to the topic of Very Large Scale Modeling. The relevance will explained in the next few paragraphs.

Software needs have managed to remain one generation ahead of the available hardware. In the context of mathematical modeling, modeling aspirations have always demanded more memory and greater speed than that which is available. However, desktop workstations with upwards of 64 MB of RAM are becoming more common place, and processor speeds, by all metrics, are higher by a factor of 5 to 10 than they were 5 years ago; the gallium arsenide chip, which is expected to raise the performance of these machines by another order of magnitude, is not far away. Parallel processing on distributed processors is also becoming available on workstation class machines.

The World Wide Web (WWW), among its yet untold effects, has brought information availability to unprecedented heights. The latest technical report from Argonne National Laboratories, or data on an arbitrary chemical species, is now available via one's favorite *web crawler*. The implications for modeling are numerous. The fidelity of a model need not be limited by information availability. The ability to share models, and tools for their solution, has only become easier. The possibility of *distributed modeling* is now real [Pon94].

Increased awareness of the need to protect the environment has led to the development of new products and services. It has also led to the need for models of higher fidelity and greater size. Chemical engineering for example has seen changes in the way that plants are designed and operated; it is no longer acceptable to pump untreated effluent down to the nearest river. As a consequence, chemical species which were once considered *trace components* now exist in process streams in much greater concentrations. The effects that these species may have on the main process will now have to be modeled. An interesting side effect in the domain of chemical engineering is the change in the *density* of the problem under consideration. The sparse matrix associated with the model of a chemical plant has roughly the same number of nonzeros per row as the number of chemical species. The writer thinks that with the need to monitor these additional species that chemical engineering matrices could see an order of magnitude increase in density over the next few years. The modeling practices and sparse matrix solution techniques will need to be rethought. This environmental protection awareness will affect other disciplines and their modeling practices in similar ways.

A competitive global economy, with the need to get higher quality products and services to the market place as early as possible, can only increase the modeling demands. Models will need to be of higher fidelity, which in most cases increases their size, complexity and connectivity. It will require that concurrent multidisciplinary modeling become the standard way of doing design. In addition the time taken to develop and to formulate these complex models has to remain low, despite their increased size and scope, due to *time to market* considerations.

The above discussion then sets the framework for the rest of this work. Models will only become larger and more complex, requiring techniques to deal with the critical issues of *speed*, *memory* and *complexity*. In light of these issues, the primary hypothesis of this work is:

*A dedicated **environment** is required to **efficiently** conduct the practice of Very Large Scale Modeling.*

In more concrete terms, the objective of this work is to provide this environment, so as to allow an order of magnitude reduction in the time taken to formulate and solve arbitrary problems with up to 250,000 variables on a reasonably equipped, currently available workstation.

Given this background, and the objective of this work, the rest of this thesis is organized as follows:

In Chapter 2, a detailed requirements analysis is done of a Very Large Scale Modeling (VLSM) Environment. The strengths and weaknesses of the state of the art systems are discussed. Finally the ASCEND III system is chosen as the platform for the start of the development of a VLSM environment.

Chapter 3 looks critically at the ASCEND III system and sets about designing a new language called ASCEND IV to fulfill the needs of a VLSM environment as laid out in Chapter 2. The emphasis here is on *scalable* language constructs, reducing the memory required for representation of large models, and features that will allow fast realization of model instances. The features needed to support fast function and gradient evaluation (a critical part of most numerical solution techniques) are also discussed. In Chapter 4, the implementation details of some of the new features of ASCEND IV are described.

Chapters 5 and 6 address the solution times of the linear algebra system, which is perhaps the most memory intensive and speed limiting feature of large scale modeling. Algorithms which make use of hierarchical decomposition in a novel way are presented.

Finally, the successes and failings of this research are summarized, along with the directions for future research.

# 1.2 REFERENCES


[Asp93] Aspen Technology, Inc. *Advent Examples Manual*, 1993.

[BA94] R. Banares-Alcantara. Design support systems for process engineering I. requirements and proposed solutions for a design process representation. Technical Report 1994-07, Dept of Chemical Engineering, Edinburgh University, Edinburgh, Scotland, 1994.

[Bar92] P. I. Barton. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, 1992.

[BKM88] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user's guide*. Scientific Press, 1988.

[CFSB94] D. J. Costello, E. S. Fraga, N. Skilling, and G. H. Ballinger. epee: A support environment for process engineering software. Technical Report 1994-19, Dept of Chemical Engineering, Edinburgh University, Edinburgh, Scotland, September 1994.

[Duf77] I. S. Duff. MA28 - a set of fortran subroutines for sparse unsymmetric linear equations. Report r8730, AERE, HMSO, London, 1977.

[FF92] P Fritzson and D. Fritzson. High-level programming support for mechanical analysis. *Computers and Structures*, 45(2):387–395, 1992.

[Gay91] D. M. Gay. Automatic differentiation of nonlinear AMPL models. Numerical Analysis Manuscript 91-05, AT&T Bell Laboratories, August 1991.

[Geo92] A. M. Geoffrion. The sml language for structured modeling: Levels 1 and 2. *Operations Research*, 40(1):38–57, 1992.

[HL88] Y.Y. Haimes and D. Li. Hierarchical multiobjective analysis for large-scale systems: Review and current status. *Automatica*, 24:53–69, 1988.

[IP95] Marianthi G. Ierapetritou and Efstratios N. Pistikopoulos. Batch plant design and operations under uncertainty. *Industrial & Engineering Chemistry Research*, 1995. Submitted and accepted.

[KFD91] D. N. Kelly, F. C. Fatora, and S. L. Davenport. Implementation of a closed loop real-time optimization system on a large scale ethylene plant. Meeting of the Instrument Society of America, Anaheim, California, October 1991.

[Las70] L. S. Lasdon. *Optimization Theory for Large Systems*. The MacMillan Company, 1970.

[LSKC93] S. N. Levy, E. Subrahamanian, S. Konda, and R. et al. Coyne. An overview of the n-dim environment. Edrc–05-65-93, Carnegie Mellon University, February 1993.

[MA91] A. Mattson and M. Andersson. Towards a universal modeling language. *ISA/91*, pages 571–578, 1991.

[MMT70] M. D. Mesavoric, D. Macko, and Y. Takahara. *Theory of Hierarchical Multilevel Systems*. New York: Academic Press, 1970.

[PB95] C. C. Pantedlides and H. I. Britt. Multipurpose processs modeling environments. volume Proc. Conf. on Foundations of Computer-Aided Process Design 94 of *CACHE Publications*, pages 128–141, 1995.

[PF89] G. E. Paules and C. A. Floudas. APROS: algorithmic development methodology for discrete-continuous optimization problems. *Operations Research*, 37:902–915, 1989.

[Pie89] P. Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[PMW93] P. Piela, R. McKelvey, and A. Westerberg. An introduction to the ASCEND modeling system: its language and interactive environment. *J. Manage. Inf. Syst.*, 9(3):91–121, Winter 1992-1993.

[Pon94] J. W. Ponton. Software environments for more effective process engineering. Technical Report 1994-23, Dept of Chemical Engineering, Edinburgh University, Edinburgh, Scotland, September 1994.

[RH93] K. Radharkrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver of Ordinary Differential Equations. Technical Report UCRL-ID-113855, Lawrence Livermore National Laboratory, December 1993. NASA Reference Publication 1327.

[spe90] *SpeedUp*, 1990.

[WHMW79] A. W. Westerberg, H. P. Hutchinson, R. L. Motard, and P. Winter. *Process Flowsheeting*. Cambridge University Press, 1979.

[Wri93] M. H. Wright. Some linear algebra issues in large-scale optimization. volume Linear Algebra for Large-Scale and Real Time Applications of *NATO Advanced Study Institute*. Kluwer, 1993.

[ZBLZ95] S. E. Zitney, L. Brull, L. Lang, and R. Zeller. Plantwide dynamic simulation on supercomputers: Modeling a bayer distillation process. Escape 95, 1995.

[Zen90] A. Zenios. Integrating network optimization capabilities into a high-level modeling language. *ACM Transactions on Mathematical Software*, (2):113–142, 1990.

# CHAPTER 2 VERY LARGE SCALE MODELING ENVIRONMENTS

## 2.1 ABSTRACT

Very Large Scale Modeling (VLSM) requires a harmonious blend of language, algorithms, and tools to be done successfully and efficiently. The sheer size of the modeling problem raises issues that are trivial when dealing with small problems. This chapter discusses the problems that need to be addressed to allow efficient formulation, representation, debugging and solving of problems with up to 250,000 nonlinear variables. The primary hypotheses is that for this undertaking to be successful, a dedicated modeling environment is required.

A number of Very High Level Languages are discussed in detail and whilst none are suitable in their current incarnations, the ASCEND III systems is seen to possess the best architecture to allow its modification to suit the needs of a VLSM environment.

## 2.2 INTRODUCTION

Very Large Scale Modeling (VLSM) is a broad, all encompassing area. It is then necessary to restrict the following discussion to models that may be represented as a set of differential-algebraic relations and the variables incident upon these relations. The relations may be equalities or inequalities whereas the variables may be continuous or discrete valued. Additionally, one or more of the variables may be required to be maximized. This restriction still admits a very large class of problems that are of interest to chemical engineers although it probably excludes most of finite element modeling[1]. Furthermore it will be assumed that the models developed will be solved by numerical techniques.

This work seeks to make modeling and solution of problems with up to 250,000 variables a routine occurrence given the current levels of computing resources and that *reasonably* skilled people are involved in the modeling process.

To achieve these goals it is necessary to examine the currently available modeling technology. The principal difficulties seem to lie in the very broad areas of:

- memory.
- speed.
- complexity.

The first two are internal concerns for the developer of an efficient modeling environment and normally manifest themselves in the inability of the end user (the modeler) to be able to model a problem of the size/complexity that he/she desires. The third affects the modeler(s) directly: how to deal with literally millions of pieces of information. If the models developed are eventually going to be solved by some numerical technique, the issues of bounding, scaling and initialization have to be addressed for *every* variable in the problem. The data gathering process to feed large models may be considerable. In some applications, such as real time optimization, this data arrives at the rate of thousands of points

1. auto gridding of fem is not considered, but representation of the problem once gridded is considered.

every 1/2 second from the plant control system. The post solution analysis of the problem is also very important.

The rest of this chapter attempts to raise an appreciation for the difficulties involved in creating a VLSM system. First, a more detailed description of the model classes that this work attempts to accommodate will be given. A requirements analysis of a VLSM environment will then be done. This analysis was made possible by having used a number of such systems. A non-exhaustive survey of such systems, their strengths and weaknesses, will follow. Finally a recommendation is made for the use of the ASCEND III system as the starting point for development of a VLSM environment.

## 2.3 MODEL CLASSIFICATION

The models that can appear within the overall model problem may widely vary. Associated with each model class are peculiarities which present challenges for solving individual models, and coordination among models becomes a formidable task.

Models may be broadly classified into black-box models and grey/glass-box models. The former group is characterized by lack of information on their structure and implementation. In general, analytic derivatives are not available and the functions involved may be discontinuous. These models are typified by *canned* subroutines that are accessible only through a subroutine prototype where inputs and outputs are specified by the model developer. When embedded within a calculational sequence, black-box models may not yield a solution and in such cases tend to do so without any *stack backtracing*, thus providing limited if any information on the cause of failure. Grey/glass-box models represent the antithesis of black-box models. They tend to be equation-based, where the functional form of the relations (equalities, inequalities, objective function) are known. The difference between grey and glass-box models is the degree to which this information is accessible. Most of the modern modeling languages such as ASCEND [PMW93] and **GAMS** [BKM88] promote a glass-box style of modeling.

The most common model classes that are found in chemical engineering design are described below. Most models may be coerced into these classes at them of solution e.g., a system of ordinary differential equations (ODE) is very often solved by discretization, resulting in a set of algebraic equations.

- Linear and non-linear equations solving of square systems of equations

$$\begin{aligned} h(x, u) &= 0 \\ u &\quad \text{given} \end{aligned} \tag{1}$$

- Ordinary differential equations (ODE) and differential algebraic systems (DAE) [RH93]

$$\begin{aligned} \dot{z} \equiv \frac{d}{d\xi}z &= f(z(\xi), u(\xi), \xi) \\ z(\xi_0) &= z_0 \quad \text{given} \\ u(\xi) &\quad \text{given} \end{aligned} \tag{2}$$

- Constrained simulation problems (CSP) [Bul91]

$$\begin{aligned} h(x, u) &= 0 \\ g(x, u) &= 0 \\ u &\quad \text{given} \end{aligned} \tag{3}$$

- Linear programming (LP) and non-linear programming (NLP)

$$\begin{aligned} & min_{x, u} f(x, u) \\ h(x, u) &= 0 \\ g(x, u) &= 0 \\ u &\quad \text{given} \end{aligned} \tag{4}$$

- Mixed integer linear (MILP and mixed integer non-linear programming (MINLP) [SG91]

**(5)**

$$min_{x,y} f(x, y) + b^T y$$

$$g(x) + Ay \le 0$$

$$x \in \Re^n$$

$$y \in \{0, 1\}$$

## 2.4 SURVEY OF MODELING LANGUAGES

Some of the state of the art modeling languages and environments will be described in the next few sections. A case will be argued for the use of a strongly typed, equation-based, object-oriented language, as the base language for a VLSM environment.

### 2.4.1 OBJECT-ORIENTED LANGUAGES

In order to motivate the discussion, consider the design of a chemical engineering process plant.

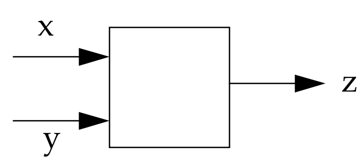



FIGURE 1 Basic model of a process plant

At an early stage of the design the model may represented as shown in Figure 1. The model is a black-box, and there is a requirement to calculate a certain quantity of material *z*, given the availability of inputs *x* and *y*. As the design proceeds the internals of the box will become more apparent; the plant is now broken down into the feed, reaction and separation sections. These can and will be further divided into the unit operations that may up each section, the reactors, columns, heat exchangers and pumps among others. Towards the final stages of the design, there will be models describing the layout of the piping and even the size of the hold down bolts for the pumps.

This process of recursively decomposing a problem into sub-parts to make it more tractable is known as *hierarchical decomposition*. For any of the sub-models there will may be an *ancestry* of models. Initially the heat exchangers may be modeled as ideal machines with infinite area; later they may be modeled rigorously using Bell's method (see the HEDH, [Sch83]). There are a number of reasons why simpler, less rigorous models are used in place of the intended final model. At a given stage of the overall design process, it is the gross features of the design that may be important and a rigorous model is unnecessary. Indeed the rigorous model may not exist, as is in the case with new products or processes. Rigorous models tend to be difficult to initialize, until they are well understood. A technique which is often used to cope with this problem is to a create a reduced model, explore its solution space, then use these results to initialize a more rigorous model. This may be thought of as *structural homotopy*.

What should be apparent is that there are two distinct and very important concepts that a modeling language must support in an efficient manner across all phases of model development: the concepts of hierarchical decomposition and refinement. When combined these techniques form the core of *object-oriented modeling.*

Object oriented techniques have become popular in the last decade. Initially developed to manage large software engineering projects [Mey88], they have found their way into database design and into modeling languages. The ASCEND III language, Omola [And90] [Nil93], gPROMS [Bar92], and MODEL.lA [SHL90] all support the object-oriented paradigm of modeling.

In ASCEND, `MODEL`s are containers for variables, relations and for other models. The *part-of* notion is achieved with the `IS_A` construct. At the leaves of the model hierarchy are the variables in the model which are called `ATOM`s. The language supports refinement and specialization through the use of its `REFINES`, and `IS_REFINED_TO` constructs, and allows any part of a model to later become more specialized through deferred binding. This applies not only to the static type information but also to instances of models. The use of an incremental compiler makes this possible.

Omola is an object-oriented modeling language built on top of an earlier structured modeling language DYMOLA. It is biased towards the modeling and simulation of dynamic chemical engineering problems. It has been used by Nilsson to model entire chemical engineering flowsheets and recently in the K2 study of a powerhouse [EN94].

gPROMS [Bar92] borrows constructs from both Omola and ASCEND and adds a number of new features. In particular it emphasizes the modeling of combined discrete/continuous processes. SpeedUp supports structured modeling but only allows hierarchies of limited depth. **GAMS** at the time of writing has no hierarchical or even modular modeling constructs and may be thought of as being *planar*. Of the languages mentioned, Omola, ASCEND and **GAMS** are *domain independent*.

### 2.4.2 TYPING

A strongly typed language has benefits certainly in the traditional object oriented languages. Its utility seems unexploited in the object oriented modeling languages. The older simulation languages which have the notion of a module with very definite functionality are strongly typed languages.

### 2.4.3 PROCEDURAL VERSUS DECLARATIVE

A model is a collection of variables and the relationship among those variables. These relationships may be conditional depending on the values of other variables. The dependency may be static, being determined at the time of realization of the model, or dynamic, being determined during the solution process.

Two main representations exist for relations among variables (hereafter simply called relations). These are the *procedural* or *functional* representation, and the *declarative* or *equation based* representation. The earlier modeling languages required a procedural representation, reflecting their heritage as being extensions of general procedural programming languages such as FORTRAN. In a procedural representation, a relation is represented as

$$y_i = g(x, u) \qquad \textbf{(6)}$$

$$i \in 1 \ldots m$$

where the sets $y$ and $x$ are disjoint, and represent output and input variables respectively, and $u$ is a vector of parameters. With this representation there is very little flexibility to change what variables are calculated. The declarative representation has no preset notion of what variables must be calculated and, in the most general representation, states what must be true at the solution of the problem. These relations are of the form

$$f(x, u) = 0 \qquad \textbf{(7)}$$

The modern languages such as **GAMS**, ASCEND, gPROMS and Omola all support the latter style of modeling. To determine what should be calculated versus what should be held fixed during the solution process, the different languages use mechanisms based on tagging the fixed variables (i.e., manipulating an attribute associated with variable). The ASCEND language takes this one step further and makes no differentiation among continuous, integral, boolean or state variables leaving the interpretation to the eventual solver and the definition that it sets for a variable. As will be seen later in this work, the utility of the extra flexibility is questionable.

The issue of a procedural versus declarative representation is not simply one of syntactic beauty. The efficiency of the solution process can be drastically affected by the insistence of which variables should be calculated. The procedural representation in most cases requires the solution of the system implied by Equation 6. However, when embedded as part of a larger problem, many sub-iteration loops may be set up, resulting in lost overall efficiency of solution. In addition, recent work by the author (unpublished manuscript) has demonstrated the dramatic changes on condition number of the jacobian matrix associated with a nonlinear system of equations, for different choices of computed variables.

In many traditional programming languages there is a formal *declaration* of variables and functions. This is normally associated with the requirement that the declaration of these declarations precede their use. This is normally done for efficiency of type checking. These traditional languages also support *explicit iterators*, through some form of *do-while* or *for* loop, for constructing repeated structures. Some workers ([Nil93], [Pie89]) have argued that these requirements and utilities imply order, which is a '*foreign construct in a declarative language*.' A pure *declarative language* in the context of mathematical modeling seems to be of little utility. However, as just seen, a *declarative statement* of the relationship among the variables in a model is often of very great utility. Particularly in the case of very large systems, the potential inefficiencies that may be incurred by an overly strict interpretation of the term *declarative* seem unjustified.

## 2.5 MEMORY

### 2.5.1 BACKGROUND

If there is one invariant feature of large scale problems, it is perhaps the great computational resources required to represent and to solve them. Many of the algorithms in the early days of mathematical programming were designed around minimizing high speed memory requirements; computers just did not have enough RAM. This restriction on memory is largely responsible for the early decomposition algorithms and sequential solution strategies including sequential modular strategies used in chemical engineering flowsheeting. The block decomposition schemes used in linear algebra [WB78] were also heavily motivated by memory limitations.

In recent years the cost of high speed memory has dropped significantly. Consequently, the minimum memory configuration (certainly for personal computers) has moved from 512K to a 8MB over the last decade. Workstations are now routinely outfitted with 64 MB of RAM, and units with 256 MB to 1 GB of RAM are becoming more common. At the time of writing, technology is on the side of the mathematical modeler. Nevertheless, the need to be concerned with efficient use of memory can not be overstressed. Aspirations of modelers tend to outstep the limits of the available hardware. The need to keep memory requirements low has often resulted in limits on problem sizes as implementors make use of short integers rather than integers for vector indexes (2 bytes versus 4 bytes on 32 bit architecture). Furthermore, some of the older languages did not support dynamic memory allocation and thus have hard wired limits on their size. Numerical accuracy has also been sacrificed, by the use of single precision versus double precision arithmetic. Finally the requirements of low memory tends to conflict with the requirements of high speed of solution. The complexity bound of an algorithm is effected by the data structures used, which has a direct affect on the memory requirements. In other cases, out of core techniques are used to augment fast memory. Reading and writing of the former is significantly more expensive.

The memory required to pose and solve a problem may be divided into the memory for the code itself, that for the data (the model), and for the solution algorithm. These will be discussed in the next few sections.

### 2.5.2 MEMORY FOR PROBLEM REPRESENTATION

If the purpose of modeling a problem is to achieve a solution vector of the calculated variables then *everything* else needed to set up and to solve the problem is *overhead*. Strictly speaking, this includes even the memory required to represent the relationship among the variables and all the features (structure, support tools and user interface) needed to deal with the complexity of the problem.

If this definition is relaxed somewhat to include the lower bounds, upper bounds and nominals (for scaling) on every variable and some allowance is made for constants, the cost of representation is $O(n)$ , where $n$ is the number of calculated variables. At 32 bytes/variable (4 double precision numbers at 8 bytes apiece (value, lower bound, upper bound, nominal)), one would require a trivial amount of memory for even $n = 1x10^5$.

However, things are not quite so idealized. The overhead associated with an object oriented paradigm is high. In addition to the minimum 32 bytes, variables need to carry around flags telling their state (fixed or free), information concerning their ancestry, and type among others. The ASCEND III supports a dimensionality field for all its variables (`ATOM`s) and their subatomic children. **GAMS** additionally supports Lagrange multipliers for its variables. This quickly adds up; in one system examined the cost of representing a variable was 200 bytes.

In addition, the relations need to be represented. In many of the high level modeling languages examined, the relations are held in form to be read by an interpreter to perform function and gradient evaluation. This form is normally a stream of tokens held in postfix or as a threaded binary tree [KLT91]. The gPROMS system uses a binary tree of terms (tokens), ASCEND IIIc ([Epp89]) uses a list of tokens held in postfix order and AMPL [Gay91] saves its relations as *opcodes* which are normally postfix streams of integer codes representing the different algebraic operators and offsets into a constant vector and variable vector. These representations are shown in Figure 2. The opcode representation is the most efficient for storage and evaluation, but is not as convenient as tree based representations for doing symbolic analysis of a relation.

.

infix: x ^ 2.5 + 3.0 - 9.0 *y

postfix: x 2.5 ^ 3.0 + 9.0 y * -

opcodes:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 6 | 1 | 3 | 6 | 2 | 4 | 6 | 3 | 2 | 2 | 7 | 5 | -1 |

variables:

| | |
|---|---|
| x | y |

constants:

| | | |
|---|---|---|
| 2.5 | 3.0 | 9.0 |

| code | meaning |
|---|---|
| 2 | variable |
| 3 | ^ |
| 4 | + |
| 5 | - |
| 6 | constant |
| 7 | * |

FIGURE 2 Relation representations

The estimated cost for naively representing 100,000 relations in the ASCEND system is over 300 MB. It is numbers like these that have discouraged the use of the equation based representation of modeling.

The memory requirements for computing derivatives will depend upon the method used for these computations and seems to involve a difficult memory/speed trade off. Finite difference gradients add little extra memory cost but may be slow and numerically unstable. Symbolic differentiation may be used, but generating the derivatives may be slow. To offset this, iterative calculations that requires gradients could generate the derivative structures at the start of the calculation which are then evaluated when required. However, the derivative structures would then have to be stored at a cost which is more expensive than the original relations. Otherwise the symbolic derivatives could be generated as needed every iteration with the resulting generation/evaluation speed penalty. Automatic Differentiation (AD) [Gri89] may be used instead of symbolic derivatives, a priori, or as needed during a calculation. However, Gay [Gay91] reports AD, though fast to generate and evaluate, can be very memory expensive. He reports on a problem where it takes over 100 MB to save the derivative structures for a *1000 variable* problem.

Finally, the structures that are needed to support the features of object-oriented and/or structured modeling also need to be represented.

### 2.5.3 MEMORY FOR SOLUTION

For large systems, the solution algorithm will determine a large portion of the memory cost. The largest cost item tends to be the cost of representing and solving the underlying linear system of equations. Even when sparse matrix technology is used (which is almost a given for large problems), the storage cost is normally given as being $k\tau$, where $\tau$ is the number of nonzeros in the *final* solution matrix and *k* is a small integer; the **ma28** manual [Duf77] puts this at around 5. When using the direct methods of solving a general linear system of equations, some variant of Gaussian elimination is used to convert the matrix into its upper and lower triangular factors. The number of nonzeros in the final matrix is not known *a priori* but will depend upon the fill in during factorization. This in turn is dependent upon the nature of the problem being solved, the factorization algorithm, the effectiveness of the sparsity preserving reorderings which are normally used in these direct methods, and the pivoting strategy.

To put the size into perspective, consider a problem with order $n$ = $1\text{x}10^5$. In chemical engineering problems, $\rho$, the average number of nonzeros per row in the initial matrix will vary from 4 to 15. As one models with more trace species, this number will only become higher. Using a $\rho$ of 10, the original matrix then has $1\text{x}10^6$ elements. The fill in may require a factor of 8 to 10, so that the number of elements in the final matrix, $\tau$, is then $1\text{x}10^7$. Using 8 byte double precision numbers $1\text{x}10^8$ bytes (100 MB) is necessary just to store the real values of the matrix after factorization. Actual memory is *k* times this size, resulting in $O\left(10^8\right)$ bytes for representing the linear system alone.

The solution algorithm will add its own overhead of vectors for its implementation. This overhead tends to be proportional to *n*, the order of the system. The cost may be nontrivial depending upon the solution algorithm being used by a solver.

## 2.6 SPEED

### 2.6.1 BACKGROUND

A VLSM environment has to be concerned with the time taken to formulate, debug and solve a problem. The existence and continued development of very high level modeling environments has been justified by the assistance that they lend to the model formulation and debugging process. Despite all the other benefits that a high level modeling language may offer over lower level representations (FORTRAN and C codes), their use will be limited if they are not within reasonable performance of the lower level languages; they will remain prototyping environments with *real modeling* left to hand crafted code. Indeed continued research in Automatic Differentiation [Gri89], which has resulted in systems that will generate derivatives for an arbitrary FORTRAN or C program, has removed one of the main, if understated, advantages of high level modeling languages; the modeler need not be concerned with the provision of derivatives.

A universal problem for modelers, independent of the implementation language or environment of their models, is that of obtaining solutions in a time which is proportional to the size of their problem. This may or may not be possible, as there are known problem classes (integer programming) where no known polynomial time solutions are known to exist. At worst case, a modeling environment should allow a modeler to access tools and algorithms easily as they become available.

The areas of speed that a high level environment has control over include the speed of realization of a model and the speed of function and gradient evaluation. These will be discussed in the next few sections. Other areas where a VLSM environment can assist in speed will be taken up in Chapter 6.

### 2.6.2 REALIZATION

The process of translating one or more input files of a model (typically an ASCII file) into the internal representation (IR) that a solver and user interface can query normally involves a number of phases. A popular model of translation is that of:

- syntax analysis.
- semantic analysis.
- code generation.

In the syntax analysis phase a lexical analyzer (scanner) and a parser work together to break up the input file first into a stream of tokens, and then to check that the sequence of tokens satisfy the grammar of the language. If a production is found to be syntactically correct, then a data structure is created that captures the essential information of the production. This process is continued until all the specified source files have been analyzed. This constitutes the *static* analysis phase. The semantic analysis phase operates upon these data structures to ensure that the data is semantically correct, after which another data structure(s) will be created which is normally a decorated syntax tree. The amount of work done at each stage is highly dependent upon the language, the tightness of the grammar, the semantics of the language, and whether the data structure created at the end of semantic analysis is the desired final product.

In languages that do code generation, such as the low level languages (C and FORTRAN), the semantic analysis phase would be followed by a machine code generation phase. SpeedUp, CSSL and many of the earlier simulation languages, as Barton points out in his thesis [Bar92], generate intermediate code, such as FORTRAN, which is then passed through another translator (a FORTRAN compiler) which would emit the machine code. He describes these languages as *code-generators*; they generate code in some intermediate language which is then compiled and linked against the solution routines to create executable simulation code.

In other languages such as ASCEND and gPROMS, the data structures which exist at the end of the syntax analysis phase represent a *type library.* In the case of the ASCEND system this phase is very fast; it takes 3 to 4 seconds to process the 15 model files necessary to represent an ethylene plant, inclusive of the thermodynamic model library files, the library of physical properties for 40 components and all the unit operations.

The semantic analysis phase is normally much longer. In this phase the model is checked for type consistency, array bounds are checked, the variables are created and eventually the relation structures are created. Most of the memory needed to represent the problem will be created in this phase.Table 1 gives the instantiation times in the ASCEND IIIc system, for a 5 component tray model using rigorous thermodynamics. (see the Appendix for a description of a tray and distillation column). The times to instantiate an entire column model with 5 components and columns with varying number of trays are also given.

In the ASCEND IIIc system, *all* the information necessary to speak to a solver can be generated as C-code. This is known as *black-box* code generation. The system thus has the capabilities of the code-generator languages described above. The generation routines can optionally generate only the function evaluation routines. These features were used to generate the C code representation for all the tray and column models in order to *simulate* the realization strategies used by some of the earlier systems. The time to compile the C-code representation of these models is also given in Table 1. In the table

- n is the number of variables, and
- nnz is the number of nonzeros in the Jacobian matrix.

The results obtained are very interesting. It is noteworthy that the C-code compilation times are significantly higher than the interpreted instantiation times for all the problems considered and not, surprisingly, show highly nonlinear behavior. Automatically generated code is notorious for breaking compilers; very few compiler writers would expect a single source file to contain 6727 functions.

The ASCEND IIIc compilation times are impressively linear, though *very* high. A native attempt to build a model with 100,000 relations of similar complexity to these examples would require over 3200 seconds to instantiate. Since modeling is a formulate, input, instantiate, solve, debug, formulate cycle, this instantiation cost would have to be incurred *each* time that a model formulation is changed.

**Table 1: Instantiation times (seconds)**

| Model | n (nnz) | ASCEND III | C no gradients | C with gradients | Ratio of compile times (gradients) |
|---|---|---|---|---|---|
| 5_comp tray | 114 (607) | 3.56 | 13.8 | 44.7 | 3.87 (12.56) |
| 11 tray column | 1567 (7715) | 55.92 | 189.0 | 700.0 | 3.37 (12.52) |
| 33 tray column | 4408 (21817) | 145.09 | 952.0 | N/A | 6.56 (N/A) |
| 51 tray column | 6727 (33355) | 277.34 | N/A | N/A | N/A |

N/A - Not attempted as compilation times excessively high

The other very high level systems such as gPROMS and Omola seem to have a very similar instantiation model as that used in the ASCEND system.

### 2.6.3 SOLUTIONS

The problem observed in the previous section is simple: the high level modeling languages, through their object oriented constructs, have placed considerable effort into the creation of libraries of reusable static objects but have *not* extended these constructs to the realization of these objects.

In other words, if it were possible to recognize that *m* objects of the same *type* are going to be needed to instantiate a model, then conceptually one need only perform semantic analysis for 1 object and then *clone* that object *m-1* times. If the cloning process is faster than performing the semantic analysis great savings in realization speed may be obtained. The ability to recognize that an object is structurally the same as another object comes about from the *strength in typing* of the language. Once a valid instance has been obtained (which is part of a larger model), it should be possible to create a *persistent* representation of this instance so that it may be used across multiple simulation runs and/or shared among multiple modelers. The potential savings in memory using such a technique are vast as well. As previously discussed the cost of representation of relation structures is a very large contributor to the total memory used by the internal representation of a high level modeling environment. The opportunity then exists to *share* relations, though variables must have their own individual memory. Revisiting the 33 tray model of the previous example, there would be only 4 unique sets of relations: those necessary to represent the condenser, reboiler and feed tray, and a single instance of the relations for the internal tray which would be shared by all of the internal trays. This may be naturally extended to even more primitive objects such as streams (chemical engineering) or beams and plates (civil engineering).

This is by no means a novel concept; it is the model that has been used in software engineering of large systems, where tools such as **make** (to track dependencies) and the use of library archives prevent the unnecessary recompilation of possibly hundreds of source files that exist in such systems. It is also the technique used for years in the modular modeling systems.

With these observations a sketch of a development program is then:

1. Formulate a model in a high level modeling language.
2. Submit it for translation.
3. If no errors are encountered (semantic or logic errors) and many instances of this type are potentially required, *clone* it, and put it into a dynamic object library.
4. Whenever an instance is required which is structurally the same as an existing prototype, clone the prototype.
5. If the model is well understood and mature, create a persistent representation of its dynamic representation taking the opportunity to exploit any speed gains, from using an alternate representation of the relations.

Given that some of these changes are possible, the hurdles that need to be crossed in achieving good translation speed include:

- efficient algorithms for copying objects.

- language constructs to allow automatic recognition of similar objects.
- lowering the cost of realizing the prototypical object to begin with.
- efficient persistent representations that will allow an instance to be saved and restored efficiently, with a possibly different underlying representation of its relations whilst maintaining the flexibility to manipulate the object, as if it were in its original form.

### 2.6.4 FUNCTION AND GRADIENT EVALUATION

Practically all numerical solution techniques make use of residuals and gradients in their solution algorithms. In the basic Newton algorithm for solving a system of nonlinear equations, at step *k* it is necessary to solve for a direction $\Delta x^k$ by solving the system of equations given by Equation 8, and taking a step in that direction as in Equation 9.

$$\Delta x^k = (-J)^{-1}\Big|_{x^k} f\Big|_{x^k} \tag{8}$$

$$x^{k+1} = x^k + \Delta x^k \tag{9}$$

Here $J$ is an $nxn$ matrix and $f$ and $\Delta x$ are $n$-vectors. In order to solve this linear system the residual vector, $f$ and the Jacobian matrix, $J$ need to computed at the current point $x^k$. This basic iteration may be repeated many times during the course of solution. Function and gradient evaluations can be expensive so it is necessary to be able to perform them efficiently.

In the past, it was difficult, time consuming and error prone to program the gradient relationships. Many systems then resorted to finite-differences which for many problems is an expensive process. Curtis, Powell and Reid [CPR74] have developed a very efficient algorithm for doing finite differences. However, the problem persists of numerical errors because of roundoff with finite differences.

The high level modeling languages such as **GAMS**, ASCEND and gPROMS have always provided these derivatives automatically thus relieving the modeler of such concerns. These systems compute their functions and gradients by using an *interpreter* which operates on an internal representation of their relations. As mentioned in Section 2.6.2 the older systems (SpeedUp, CSSL) tend do their function and gradient evaluation by making calls on externally *compiled* machine code.

Barton, in his thesis, bemoans the slow code compilation step of SpeedUp especially for use in an interactive environment and argues that although interpreted code runs slower than machine code, overall modeling productivity is greater with the interpreted evaluators. The numbers presented in Table 1 clearly tell how slow the code compilation step can be.

**Table 2: Evaluation times (seconds)**

| Model | ASCEND III functions | gradients | C code functions | gradients | Ratio of function times |
|---|---|---|---|---|---|
| 5_comp tray | 0.14 | 0.23 | 0.01 | 0.03 | 14.0 |
| 10 tray column | 1.63 | 3.54 | 0.07 | 0.17 | 23.23 |
| 33 tray column | 4.63 | 9.97 | 0.15 | N/A | 30.86 |
| 51 tray column | 7.05 | 15.25 | N/A | N/A | N/A |

N/A - Not attempted as compilation times excessively high

However, Table 2 shows that the evaluation times for the compiled C code is an order of magnitude faster than the interpreted code. Gay [Gay91] has reported a range of 8 to 30 times speed up for function and gradient evaluation using compiled rather than interpreted code. Extrapolating the above data to $n = 100{,}000$ would yield evaluation times of 105 versus 4.5 seconds for the interpreted and compiled code respectively. A crude optimization exercise yields a break even point (for this example) after 85 function evaluations. For the larger problems, it would seem that a choice does not exist; interpreted code *has* to be used, as it is impossible to compile the C code generated.

### 2.6.5 SOLUTIONS

Fortunately the above analysis is flawed. The C code that was generated and the resulting object file obtained from compilation is *persistent*. It may be used across multiple simulations in one modeling session, as well as across multiple sessions. This will be so until its description in the modeling language changes. The economics then looks very attractive compared to interpreted code after a few invocations.

The problem of not being able to compile the C code because of its size still remains. This again is misleading. The C code that was generated for these experiments was not done intelligently; code was generated for an entire distillation column. But consistent with the strategy of hierarchical decomposition, the code should have been generated for a single tray. Additional code should then be generated for the equations that wire up the trays. Thus the same approach for achieving instantiation speed can be used for obtaining efficient function and gradient evaluation, i.e. the use of persistent representations and recognizing repeated structures. The challenge lies in making these faster evaluation routines (but now external) accessible to the modeling environment.

A VLSM environment has considerable flexibility in determining the format of the persistent representation used. This will examined in subsequent sections.

## 2.7 MODELING SUPPORT

### 2.7.1 INTRODUCTION

The modeling activity iteratively involves posing a problem, attempting to solve it and then querying the results. Throughout this cycle, the modeling environment needs to give support to the modeler. This is true for any model but becomes particularly important for large scale modeling. In a large model posed in an equation based manner, the information that it is available and that needs to be manipulated quickly becomes overwhelming. Most modeling environments (rather than modeling language) give some form of user support. The ASCEND and Omola modeling environments are described in the Appendix.

### 2.7.2 QUERY FACILITIES

A feature that seems to be lacking in the environments reviewed is a full suite of query facilities. The ASCEND III and Omola environments are reflexive to different degrees, allowing query of their state. An instance of a model may then be considered as an *in-core* database and the user interface is really just a mechanism for probing and manipulating that database. These databases may be large. The ASCEND III system typically shows a factor of 10 objects for every for variable. It is expected that 2.5 to 3 million objects will have to be manipulated for the size of problems being targeted.

Given the volume of information embodied in a large model, a full-blown query language, perhaps based on SQL[2] should be available. This would allow queries for debugging, initialization and reporting. The ability to make queries such as

> *find all instances of vapor flows that are* < *20.0 moles/sec and are within 0.0001% of their upperbound.*

is an invaluable tool for determining why a model is suffering problems with convergence. In both the ASCEND III and Omola environments, manually browsing an instance to find information is a painful and costly exercise for large models. In addition, it should be possible to make queries across different runs of a numerical solution and indeed across different modeling sessions.

### 2.7.3 PERSISTENCY

In previous sections the benefits of persistency were discussed as a means of saving instantiation time, of increasing function and gradient evaluation speed (through C code and the persistency of the associated machine code), and of making the associated savings in memory. This requires that there be efficient tools in a VLSM environment for saving and restoring instances. These tools need to be able to ensure consistency between the static type information from which the model was derived, any external code and the persistent object. Consistent with the rest of this work, the emphasis is on efficiency.

2. SQL is the defacto standard for database query languages.

The ability to make queries across multiple modeling sessions (see Section 2.7.2) for changes in data and model structure results in a database design problem. The data representation and data base management system for a VLSM environment should be chosen with the following consideration: a VLSM environment will eventually be just one of many nodes in a wider *design-in-the-large* environment, such as that being realized through the epee [CFSB94], and *n*-dim [ndg95] projects.

### 2.7.4 TOOL EVOLUTION

Associated with model evolution is the need for tool evolution. A motivating example is the case of a given subproblem being a linear program representing an aggregated model of a heat exchanger network. At a more detailed design stage this model becomes a nonlinear program (NLP). A solver (tool) capable of handling NLPs is then necessary. Even for a fixed model type, there needs to be flexibility to change the tool used. It is a fairly common occurrence in solving optimization problems for example to experience failure using one solver and to invoke another solver and have convergence. The software engineering issues of dealing with multiple and possibly proprietary solvers is one that seems to have been successfully handled by **GAMS**. At the time of writing at least 14 different solvers are accessible to the **GAMS** modeling language ([Cor95a]).

### 2.7.5 EXTERNAL PACKAGES

For large scale modeling many models may exist in well tested *external subroutines* which may be black-box or glass-box in nature. A VLSM environment needs to be able to give seamless support to these external models and, in general, support models covering the entire spectrum of model transparency. This requires that a VLSM environment have the ability to imbed *procedural models* within a declarative environment. This has been extensively discussed by Barton [Bar92][, though he only considers the black-box case. A VLSM must also support arbitrary external function calls to allow access to code that is perhaps more efficiently written in another language, as well as for providing access to databases and other service routines.

### 2.7.6 OTHER USEFUL TOOLS

In addition to the features supported by the current state of the art modeling environments, other very useful features include techniques that run through mathematical modeling. A non-exhaustive list of these features include:

- Linearizations of relations with checks for unboundedness and linear dependence.
- Quadratic (general nonlinear) approximations to replace highly nonlinear functions.
- Algorithms for finding initial feasible starting points.
- Sensitivity analysis of the objective and constraints.
- Algorithms for analyzing degrees of freedom.
- Constraint relaxation.
- Deletion of rows and columns from a model.
- Provision of partial derivatives as a modeling construct.
- Analysis of bounds and scaling of variables and relations.
- Problem reformulation to make solution more tractable for different algorithms.

Very few of these features are implemented/accessible in the current modeling environments. Greenberg, [Gre95] among others, have examined the pre and post analysis of linear programs in some detail. The results of his research has been realized through the ANALYZE tool.

## 2.8 OTHER

The user interface issues of presenting the results of a large-scale computation or analysis to a user needs to be addressed in a VLSM environment. For example, in one system examined, a degree of freedom analysis tool for nonlinear systems of equations reported that a model was underspecified. A list of over 5,000 possible variables was then presented for the user, to fix one (1) of these variables. Obviously some information filtering needs to be done with large-scale problems.

An interactive modeling environment is invaluable in the early stages of model development. It becomes less useful as the model becomes more mature, at which time it may be safely and more conveniently run as a batch or semi-batch job. There should exist a sufficiently powerful executive language to provide control for iterative strategies and for solving problems with long execution times.

Finally, recent advances in computer science techniques such as dynamic loading and exploitation of parallel architectures should not be forgotten in the design and implementation of the ideal VLSM environment.

## 2.9 DISCUSSION

The ease of model reuse, interchange and upgrade that comes from the object-oriented programming paradigm, the numerical efficiencies that can be achieved through an equation-based format, and the potential benefits that arise from recognizing an object by its signature, suggests that the ideal VLSM environment should be based on a *strongly typed*, *equation-based*, *object-oriented* modeling language. The very serious problems related to memory, speed and complexity were discussed and solutions proposed. These solutions exploit repeated structures and make use of persistency. It has been argued that the very high level modeling languages and environments, with appropriate modifications and support tools, can prove to be just as efficient and significantly more flexible than lower level representations. The relationship between large scale (computational) modeling and design-in-the-large was elucidated.

The ASCEND III system will used as the basis for design and implementation of a VLSM environment. The decision to use the ASCEND III system as the platform is driven by its availability in the public domain, its domain independence and the conceptual cleanliness of its design. In the next chapter, a more detailed examination will be done of the issues involved in the creation of ASCEND IV, a language and environment to efficiently conduct very large scale modeling.

In closing, an intriguing question is whether a VLSM environment can *assist* in keeping the solution cost to be linear with the size of the problem being solved. Conceivably information concerning the nature of the problem being solved could be passed onto the solver. At least one language has introduced language constructs to aid this process. The **GAMS** note on *special ordered sets* for mixed integer programs, [Cor95b] describes a SOS1 set as a set where at most one member of the set can have a non-zero value. They also describe a type SOS2 and features that allow specification of special branching rules. Apparently this information is passed to solvers that may choose to ignore it if they so desire. The note points out that the introduction of special ordered sets has been driven '*...by internal algorithmic consideration, than by broader modelling concepts*'. Integer programming is a difficult task; any information about the problem is then useful. Arguably one could derive an algorithm that would find these special ordered sets. The writer doubts that it will be as fast as the $O(1)$ algorithm implied by the SOS1 construct, i.e., explicit labelling of these sets by the user.

An interesting application of a high level language in aiding the solution of large-scale linear algebra problems will be discussed in Chapters 5 and 6.

## 2.10 APPENDIX

### 2.10.1 ASCEND III MODELING ENVIRONMENT

The ASCEND III [3]system has a multi window graphical user interface (GUI) which allows interactive setting up of a problem, changing of values, and flags on variables and relations. There is a LIBRARY which gives the user access to the static representation of the model and the type ancestry of all the models in the library. Once a model has been realized, the main access is through the BROWSER. The BROWSER in ASCEND III gives a 2 level view of the model, a

3. ASCEND or ASCEND III refers to the modeling environment implemented by Piela as part of his Ph.D. thesis and later whilst doing post-doctoral work at Carnegie Mellon University up to 1992.

parent model and its children thus showing a slice of the model hierarchy. It has facilities for finding objects by name and by type but is limited to single queries. The environment supports a PROBE, where arbitrary but selected parts of a model may be *exported* and viewed. For long running and repeated exercises, it allows instructions to be given in a SCRIPT. The SCRIPT is essentially the language of the interface, and in ASCEND III is weak, not permitting any looping constructs. The SCRIPT possesses a useful recording feature which allows capture of many of the commands given through the graphical user interface. The notable features of the ASCEND environment are its interactive refinement and its SOLVER interface. From the BROWSER, a user can make any part of an model more specialized, provided that a valid subclass (in the ancestral sense) exists in the LIBRARY. The SOLVER supports automatic degrees of freedom analysis, checks for structural and numeric dependency, an incidence matrix viewer, and a DEBUGGER for querying the variables and relations currently held by the solver. The system also has software bridges to external plotters and spreadsheets and some features for saving and restoring objects.

### 2.10.2 Omola MODELING ENVIRONMENT

Omsim is an interactive environment with a graphical user interface (GUI) for defining and simulating dynamical models, based on the Omola modeling language. It contains the Omola parser which is used to load Omola model definitions into the environment. The class browser is the main window in the environment from which other tools may be activated. The model inheritance hierarchy and object hierarchy can be displayed graphically. There is also a graphical model editor. The Omsim environment has a number of built-in numerical integration routines for simulating a model. The simulator has subtools to access and to display variables, parameters and simulation results along with debugging tools.

In addition, there are features for displaying plots and saving and restoring simulations. The Omsim environment supports a command language (OCL) for performing batch operations.

### 2.10.3 A DISTILLATION COLUMN

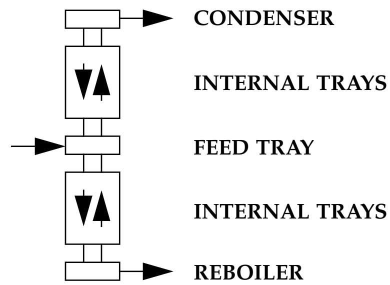



FIGURE 3 Schematic of a distillation column

## 2.11 REFERENCES


[And90] M. Andersson. *Omola An Object-Oriented Language for Model Representation*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1990.

[Bar92] P. I. Barton. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, 1992.

[BKM88] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user's guide*. Scientific Press, 1988.

[Bul91] L. Bullard. *Iterated linear programming strategies for constrained and nonsmooth simulation*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, 1991.

[CFSB94] D. J. Costello, E. S. Fraga, N. Skilling, and G. H. Ballinger. epee: A support environment for process engineering software. Technical Report 1994-19, Dept of Chemical Engineering, Edinburgh University, Edinburgh, Scotland, September 1994.

[Cor95a] GAMS Development Corp. GAMS solvers. http://www.gams.com/solvers.html, November 1995.

[Cor95b] GAMS Development Corp. Special mip features. http://www.gams.com/docs/mipfea.htm#semiv, October 1995.

[CPR74] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *J. Inst. Maths. Applics.*, 13:117–120, 1974.

[Duf77] I. S. Duff. MA28 - a set of fortran subroutines for sparse unsymmetric linear equations. Report r8730, AERE, HMSO, London, 1977.

[EN94] J. Eborn and B. Nilsson. Object-oriented modelling and simulation of a power plant. application study in the K2 project. Technical Report ISRN LUTFD2/TFRT–7527–SE, Department of Automatic Control, Lund Institute of Technology, December 1994 1994.

[Epp89] T. G. Epperly. Implementation of an ASCEND interpreter. Technical report, Engineering Design Research Center, Carnegie Mellon University, 1989.

[Gay91] D. M. Gay. Automatic differentiation of nonlinear AMPL models. Numerical Analysis Manuscript 91-05, AT&T Bell Laboratories, August 1991.

[Gre95] H. J. Greenberg. Analyze bibliography. WWW, 1995. http://www-math.cudenver.edu/ hgreenbe/analref.html.

[Gri89] A. Griewank. On automatic differentiation. *Mathematical Programming: Recent Developments and Applications*, pages 83–108, 1989. Also appeared as Preprint MCS-P10-1088, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne Ill., October 1988.

[KLT91] R. L. Kruse, B. P. Leung, and C. L. Tondo. *Data structures and program design in C*. Prentice Hall, 1991.

[Mey88] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[ndg95] The n-dim group. n-dim - an environment for realizing computer supported collaboration in design work. Technical Report EDRC 05-93-95, The Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA 15213, 1995.

[Nil93] B. Nilsson. *Object-Oriented Modeling of Chemical Processes*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[Pie89] P. Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[PMW93] P. Piela, R. McKelvey, and A. Westerberg. An introduction to the ASCEND modeling system: its language and interactive environment. *J. Manage. Inf. Syst.*, 9(3):91–121, Winter 1992-1993.

[RH93] K. Radharkrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver of Ordinary Differential Equations. Technical Report UCRL-ID-113855, Lawrence Livermore National Laboratory, December 1993. NASA Reference Publication 1327.

[Sch83] E. U. Schundler. *Heat Exchanger Design Handbook*. International Center for HEAT and Mass Transfer. Washington : Hemisphere Pub. Corp., 1983.

[SG91] N. V. Sahinidis and I. E. Grossmann. Convergence properties of generalized Benders decomposition. *Computers Chem. Engng*, 15(7):481–491, 1991.

[SHL90] G. Stephanopoulus, G. Henning, and H. Leone. MODEL.LA. A Modeling Language for Process Engineering. Part I: The Formal Framework. *Comput. chem. Engng*, 14:813–819, 1990.

[WB78] A. W. Westerberg and T. J. Berna. Decomposition of very large-scale newton-raphson based flowsheeting problems. *Computers and Chemical Engineering*, 2:61–63, 1978.

# CHAPTER 3 DESIGN OF ASCEND IV

## 3.1 ABSTRACT

Using the ASCEND IIIc language as a starting point a new language ASCEND IV is proposed. This language will form the core of a very large scale modeling (VLSM) environment. In designing the new language emphasis was placed on efficiency of representation, speed of model realization, and efficient function and gradient evaluation.

## 3.2 INTRODUCTION

ASCEND III is a modeling environment composed of a language interpreter, a solver toolbox and multi-window graphical user interface. It is built on top of the ASCEND III equation-based, object-oriented modeling language. A detailed description of the language and environment is given by Piela [PMW93], [PEWW91]. A detailed presentation of current modeling languages inclusive of

ASCEND III was given in Chapter 2. The ASCEND III language has deficiencies which make it unsuitable as a language capable of supporting a VLSM environment.

The first major problem with ASCEND III is its implementation language. It is written in the Domain Pascal (DP) language. DP is based on Pascal but with many language extensions. Compilers for DP are rare. The upshot is that ASCEND III is not portable. It was envisioned that portability would be an important feature of a VLSM environment, especially if distributed modeling would eventually need to be supported. The language compiler was rewritten in C by Thomas Epperly. The solver was rewritten in C, by Karl Westerberg. Using the C versions of the code another graduate student[1], Benjamin Allan, and the author rewrote the graphical user interface (GUI) and pulled the entire system together over a 9 month period. The GUI was written in Tcl/Tk [Ost94] and the opportunity was seized to *open* the system. Every aspect of the system is now modifiable by the user, though good defaults are provided. The scripting language is now disguised Tcl, which gives access to the GUI, the internal ASCEND III datastructures and the underlying operating system. The system is now known as ASCEND IIIc; wherever necessary, a distinction will be made between ASCEND III and ASCEND IIIc.

Given the requirements analysis of a VLSM environment, as discussed in Chapter 2, the following deficiencies were also found with ASCEND III:

- closed interface (fixed in ASCEND IIIc).
- no input/output capabilities.
- poor connectivity/connection to its solvers.
- weak procedural capabilities, both internally and externally.
- inability to talk to external code.
- slow instantiation of models.
- slow solution speed for large models.
- slow procedure execution.
- very high memory requirements for representation of objects, in particular relations.
- little support for model reuse.

1. and very good friend of the author

This has prompted the design of a new language/environment which will be known as ASCEND IV. Unless otherwise specified the term ASCEND IVwill used to describe the language and/or the environment. The term ASCEND IV.alpha will be used to describe the prototype implementation of ASCEND IV used during this work.

The following statement is taken from the synopsis of Piela's dissertation:

> *"Our hypothesis has been that a special purpose modeling language will not only reduce the time it takes to build these systems (models), but will also give designers a formalism by which they can organize and share their work in a cooperative manner"*

ASCEND IV seeks to build on this premise in a highly efficient manner and uses the terms Very Large Scale as its guiding adjective. This means scalability. ASCEND IV has to be as much like ASCEND III as possible, and more. However whenever conflicts of flexibility versus efficiency arise, the new language errs on the side of the latter. The rest of this chapter is organized as follows:

The ASCEND IIIc language will be described in some detail. Using examples, the deficiencies of the language will highlighted and solutions will be proposed. Finally conclusions about the new language, ASCEND IV, will made.

## 3.3 ASCEND IIIc

In order to keep the discussion relatively self contained, the critical features of the ASCEND III language will be discussed. The language supports `MODEL`s, `ATOM`s, and the fundamental types of **real**, **integer**, **boolean**, **symbol** and **set**. The type system is rooted on these fundamental types. `ATOM`s all inherit from one of these base types. The distinctive feature of `ATOM`s is that they have may have a value. In the case of real `ATOM`s, there is dimensionality associated with the value. `MODEL`s are made up of `ATOM`s, other `MODEL`s and relations. Though strictly not classified as `ATOM`s, relations behave like `ATOM`s and have a value (residual) and dimensionality. `MODEL`s and `ATOM`s may be made more specialized through inheritance or refinement using the REFINES construct. The semantics implied by inheritance is one of textual substitution. A type may inherit from only one type, i.e. single inheritance. New parts (children, slots, fields, parts are all synonymous terms) may be added, but none may be removed in a refinement to a more specialized type.

Within the scope of a MODEL definition there is a declarative section and a procedural section. The start of the procedural section is given by the INITIALIZATION keyword. In the declarative section, variables and relations are declared and constructed. The four main constructs are, IS_A, IS_REFINED_TO, ARE_THE_SAME, and ARE_ALIKE.

The IS_A construct serves to declare an instance of a type and, at the time of instantiation, becomes the constructor for the type. In effect the IS_A behaves like the automatic variables in a language such as C or FORTRAN. For instances of MODELs, the construction takes place recursively. The analogue in C is that of structures, having slots which are themselves structures, *but* containing no pointers. Only value and not reference, semantics is applied.

The ARE_THE_SAME construct allows the merging of two or more instances of compatible type. The merge is recursive and, in the simplest case of merging two ATOMs, will yield a single instance of an ATOM which may be referred to by two (2) different names. For both MODELs and ATOMs, assuming that type compatibility exists, the more refined MODEL/ATOM will exist at the end of the merge. The checking of the type compatibility has to be done recursively for models.

IS_REFINED_TO allows any instance to be transformed into a more specialized type, provided that the new type is *type compatible* with the old type. IS_REFINED_TO may be applied to any part of a MODEL but not to parts of an atom. ASCEND IIIc introduced this restriction to allow more efficient representation of ATOMs.

The ARE_ALIKE construct is used for type propagation. If a number of instances are said to be ARE_ALIKE, then changes in the type of any one of the instances in the *clique* of instances will be propagated to all the members. Change in the type of an instance is achieved explicitly through the IS_REFINED_TO construct, and both implicitly and explicitly through the ARE_THE_SAME operator.

In the procedural section, PROCEDUREs are used for doing assignments of values to ATOMs. The standard iterators that are found in most procedural languages are provided. PROCEDUREs are not however parameterized and operate within the scope of the instance to which they are bound. Procedures may also call other procedures at the same scope.

## 3.4 PPP EXAMPLE

The example described here is that of a simple separation sequence involving 3 components and 2 distillation columns. The components (i.e. the chemical species) are propylene, propane and propadiene and represent the C3 separation section of a hydrocarbon process.

**Table 3: Instantiation Statistics**

| | ASCEND III |
|---|---|
| # Relations | 17516 |
| # Variables | 18113 |
| Size [1](MB) | 65.36 |
| # Instances | 247380 |
| # Formal Types | 43 |
| # Arrays | 13581 |
| Instantiation Time [2](sec) | 985.76 |
| Function Time (sec) | 13.19 |
| Gradient Time (sec) | 28.73 |

1. Memory does not include system overhead of 5.74 MB

Throughout this document this example will be referred to as the example PPP. A sketch of the process is given in the Appendix. Rigorous thermodynamic models are used throughout. The number of trays in the first and second columns are 30 and 164 respectively.The performance of ASCEND IIIc is shown in Table 3. As before, scaling to 100,000 relations assuming linear behavior, the instantiation time, and memory requirements would be over 5700 seconds and 373 MB respectively.

### 3.4.1 MODEL INSTANTIATION

Instantiation in ASCEND III is slow, as seen from the above results. There are numerous reasons for this, and it is perhaps easiest to understand these reasons though an example. In the following valid code fragment, the `MODEL` **test1** introduces a number of streams and also specifies that *stream[1]* is a **liquid_stream**, whereas the other streams are **vapor_stream**s. `MODEL` **test2** makes use of **test1** and specifies the number of streams that are to be created and the types of their constituents, in this case, acetone, benzene and chloroform. `MODEL` **test3** makes **test2** more specialized and adds an *interface* to the problem by the `ARE_THE_SAME` on line 24. If an instance (named *t3*) of **test3** were to be created, then one could now say *t3.T* rather than *t3.stream[1].T*.

```
1  MODEL test1;
2
3       nstreams IS_A integer;
4       stream[1..nstreams] IS_A td_stream;
5       stream[1] IS_REFINED_TO liquid_stream;
6       stream[2..nstreams] IS_REFINED_TO vapor_stream;
7
8  END test1;
9
10 MODEL test2;
11
12      components IS_A set OF symbol;
13      t1 IS_A test1;
14      t1.stream[1..nstreams].components,
15           components ARE_THE_SAME;
16      components := [‘acetone’,’benzene’,’chloroform’];
17      t1.nstreams := 4;
```

```
18
19 END test2;
20
21 MODEL test3 REFINES test2;
22
23      T IS_A temperature;
24      T, t1.stream[1].T ARE_THE_SAME;
25
26 END test3;
```

EXAMPLE 10 Ascend Code Fragment

Of the three models shown here, **test2** and **test3** have complete definitions and may be said to be *closed*. **test1** requires information about the number of streams (*nstreams*) and the types of the components for each of the streams. These are specified on line 17 and line 16 respectively.

This example demonstrates a number of problems with the language. In particular it demonstrates

1. absence of a structural parameterization.
2. lack of order.

Each of these and how they affect instantiation speed will be discussed below.

#### 3.4.1.1 OBJECT INTERFACES

ASCEND III has does not have an *interface* to any of its `MODEL`s/`ATOM`s; they must be all reached by their parents in a *part-of* sense, through the use of qualified naming. This applies to all variables inclusive of those that Nilsson, [EN94] refers to as *structure_parameters*, which are the variables necessary to define the number and size of the objects that are to be created.

The reasons for having an *interface* to an object are numerous. It is now widely accepted that interfaces provide an abstraction boundary and lies at the core of good software engineering practice [Mey88]. With an interface to an object, the underlying implementation may change, but all code depending on this object need not. Some languages (C++, Modula-3) have the notion of *private* parts which are invisible to the outside world. This notion of private parts at first seems as an attractive way to deal with the complexity of modeling large complex systems. For example, in a chemical engineering flowsheet, one is not always interested in knowing the pressure drop across every valve in every section of a plant. However, as Piela argues in his thesis [Pie89], in an equation based system, *any* variable may be calculated; it is then difficult to determine what a good interface should be. Nilsson also discusses these issues at length in Chapter 6 of his thesis [Nil93].

#### 3.4.1.2 STRUCTURAL PARAMETERIZATION

The lack of structural parameterization requires that the values for these structural variables must be set by deference or *lifting*. This was seen on line 17 of Example 10. In trying to create an instance of `MODEL` **test2**, the instantiator starts the construction of part *t1* but has to stop as it does not know the value of the structural variable *nstreams*. It then gets this value and can proceed.

This is possible through the use of an incremental compiler that uses *lazy evaluation* and multipass instantiation [Epp89]. The basic principle is that a pending queue is used, and partially instantiated objects are placed on the pending queue. The mechanism used to keep track of the completeness of an object is a bitlist associated with the object. There is a bit associated with each statement in the type description of the model. Each bit then corresponds to the binary states of executed or pending. The pending queue is repeatedly processed until it is determined that no more statements can be executed or that instantiation has been successful. Many *failed* attempts occur whilst trying to instantiate a model. This was just demonstrated above. A statement has to be marked as pending once it cannot be fully executed. Until all the instances necessary to execute the statement have been constructed, the execution has to be deferred. This becomes particularly expensive whilst attempting to instantiate compound statements such as FOR statements and relations. In this case, only 1 bit is associated with each statement. Relations are perhaps even more expensive to instantiate. Instantiation of a relation can not be completed until all the variables incident upon the relation have been constructed. In many cases the semantic analysis is almost complete before it is realized that some of the variables do not yet exist. The relation statement has to be marked as pending and the entire process repeated the next time that this instance/statement is visited.

#### 3.4.1.3 ORDER

ASCEND III, in addition to being equation-based, attempts to be a *declarative language*. Consequently, the statements may be written in any order. Variables may be used before they are declared, and declarations may be mixed freely with other statements. The instantiation algorithm has to respect the lack of order as provided in the language definition, so there is no guarantee of the order of which statements may be executed.

This results in a very flexible language where decisions concerning the problem to be modeled may be deferred and, in the limiting case, not even specified at the time of instantiation.

Apart from placing some restrictions on the expressiveness of the language (ordered sets, type-of, size-of queries etc.), the lack of any guaranteed order places severe demands upon the instantiation algorithm as seen in Section 3.4.1.2. In addition, the lack of order makes it very difficult to exploit repeated structures. From the data shown in Table 3, there were 13581 array instances in the final instantiated object, suggesting that large potential exists for exploiting repeated structures (see Section 2.6.3 and Section 2.6.5). Other impediments to exploiting repeated structure will seen in subsequent sections.

#### 3.4.1.4 CONSTRUCTORS

The language does not separate declaration and construction of objects. In order to achieve consistent naming ASCEND III requires the combined use of `IS_A` and `IS_REFINED_TO` constructs, which eventually proves expensive. This is demonstrated in `MODEL` **test1**. In order to provide consistent naming, an array of stream instances is created, and the necessary instance refinements is done to a achieve the required stream type. The alternative would have required:

```
MODEL test1_alternative;

     nstreams IS_A integer;
     stream_1 IS_A liquid_stream;
     stream[2..nstreams] IS_A vapor_stream;

END test1_alternative;
```

It is more expensive to instantiate an `IS_REFINED_TO` statement than an `IS_A` statement. The other reasons for separating declaration from construction will be seen later.

It should be pointed out that this style of modeling is very desirable in some situations, especially when developing generic models where the final types of the instances are not known. A typical use is shown below.

```
MODEL stream_collection;

     nstreams IS_A integer;
     stream[1..nstreams] IS_A td_stream;

END stream_collection;

MODEL liquids REFINES stream_collection;

     stream[1..nstreams] IS_REFINED_TO liquid_stream;

END liquids;
```

#### 3.4.1.5 ALIASING

In the absence of constructs that allow an object to have an interface, modelers in ASCEND III seem to have developed their own crude interfacing techniques. Referring again to Example 10, on line 23 and line 24, a new variable *T* is introduced which is `ARE_THE_SAME`'d with a variable deeper down in the model. Ostensibly the modeler would then always refer to the variable by its shorter name, thus providing an interface to the model **test3** (and a firewall to changes that make take place in the definition of *t1.stream[1].T*) .

The ARE_THE_SAME operator then serves at least two purposes in the ASCEND III language. The first is aliasing to provide an interface to objects; the second is that which it was originally intended to be used for, the merging of objects.

The overloading of the ARE_THE_SAME operator can have a very detrimental effect of the cost of instantiation. Two or more possibly very complex objects are created, only to then destroy one of them during an ARE_THE_SAME. The provision of a true alias feature, combined with separation of type declaration and construction, will remove this waste. It is interesting to note that Barton [Bar92] rejects the ARE_THE_SAME feature in the design of gPROMS, and introduces an IS construct in its place.

### 3.4.2 IMPLICIT TYPES

Object-oriented languages fall broadly into two groups: the class based systems and the prototyped based ones. A very clear exposition is given in [US91] on the differences between these two approaches. ASCEND III is a hybrid language which uses constructs from both paradigms, but, in attempting to be strongly typed, it behaves more like a class based system. The language, however, does permit *implicit typing*. This breaks the strict boundaries normally imposed in class based systems, but this is part of the powerful expressiveness of the language.

```
MODEL test1;
     var IS_A solver_var;
END test1;

MODEL test2;
     t1, t2 IS_A test1;
     t1.var IS_REFINED_TO molar_rate;
     t2.var IS_REFINED_TO temperature;
END test2;

MODEL test_all REFINES test2;
     t1, t2 ARE_THE_SAME;
END test_all;
```

EXAMPLE 11 Implicit Types

It is perhaps easiest to explain implicit types through an example. In Example 11, all of the types shown are complete. In **test2,** the parts *t1*, and *t2* are introduced as being instances of type **test1**. However, in the proceeding lines (line 7 and line 8), the part *var* in *t1* is refined to a **molar_rate** while its counterpart in *t2* becomes a **temperature**. Clearly *t1* and *t2* are now type incompatible, and any attempt to create an instance of **test_all** will fail. The modeler has just created implicit types **test1'** and **test1"**, now shown explicitly below:

```
MODEL test1';
     var IS_A molar_rate;
END test1';

MODEL test1'';
     var IS_A temperature;
END test1'';
```

The interest here in implicit types is driven solely by the impact that it has on instantiation speed and cost of representation. As suggested in Chapter 2, a significant benefit can be derived from knowing the type of object under consideration. For example, if a group of relations (as collected under a type description) is required and an *exact* group of relations has already been created, then the new relations do not need to be created, and hence they do not need to be represented. The first results in a saving of instantiation time, the second in a saving in memory. *This is the perhaps the single most important feature in the efficiency of creation and representation of models in low level languages or in traditional modeling languages employing the sequential modular paradigm.*

The difficulty described is not unique to ASCEND III. In the prototype based object oriented languages, the lack of a type system has resulted in slow execution of procedure calls. Research to address these difficulties is very active and good progress has been made with the SELF language [CU91].

A few definitions will be introduced to aid later discussions.

**DEFINITION:** A *formal* or *explicit* type is any type that was explicitly declared and defined in a type library.

The natural converse is then

**DEFINITION**: *Implicit* typing results in the creation of new types which were not explicitly declared or defined in a type library.

**DEFINITION**: A *complete* type is a formal type which may fully instantiated without further specification of structural parameters.

Implicit types are created by

- The `IS_REFINED_TO` of a child of an instance, potentially creates implicit types for all instances on the path back to the root instance.
- The `ARE_THE_SAME`ing of a formal type with an implicit type.

Likewise all incomplete types are potentially implicit types.

**DEFINITION**: Two instances created with the same formal type description and the same structural arguments are said to be *equivalent*.

This equivalence is maintained until either of the instances has its type description changed through instance refinement (`IS_REFINED_TO`) or through merging with an instance of a different type. This may or may not create implicit types. Needless to say, the user interface of ASCEND III can be the largest source of implicit types. In Chapter 2, the use of structural homotopy was described as a way of getting difficult models to solve. This structural homotopy is largely possible through the use of implicit types and has been shown to be one of the most useful features of the language.

The challenge of providing a language that caters to the development and understanding of immature models but is efficient for formulation and solution of very large scale problems is taken up in the next section.

## 3.5 ASCEND IV

In order to overcome the short comings of ASCEND III, a number of new constructs are introduced into the language. Not surprisingly, they attempt to address the items highlighted in Section 3.4. These constructs are:

- parameterized types.
- separation of declaration and construction of instances.
- ordered instantiation.
- `IS_A_PROTOTYPE`, `CLONES` and copy semantics.
- `IS` and reference semantics.

The above items are tightly coupled and are perhaps best explained by working through the earlier example but with the new syntax and semantics. A detailed discussion of the semantics will not be attempted. At the time of writing the full implications are not all known (though they have been tested in prototype implementations).

### 3.5.1 EXAMPLE 10 REVISITED

```
MODEL test1(nstreams : integer;
            components : set OF symbol);

     stream[1..nstreams] : td_stream;
     stream[1] IS_A liquid_stream(components);
     stream[2..nstreams] IS_A vapor_stream(components);

END test1;

MODEL test2;

     nstreams IS_A integer;
     nstreams := 4;
     components IS_A set OF symbol;
     components := [‘acetone’,’benzene’,’chlorofrom’]

     t1 IS_A test1(nstreams, components);

END test2;

MODEL test3 REFINES test2;
```

```
22
23      T : temperature;
24      T IS t1.stream[1].T;
25
26 END test3;
```

EXAMPLE 12 New Language constructs

MODEL **test1** is now a parameterized type over the structural variables *nstreams*, and *components*. The *:* syntax declares the type of the formal arguments to allow type checking. The first statement in this model declares an array of streams rooted at the base type **td_stream** but does *not* attempt construction of the instance(s). This step is critical with parameterized types. In the case of arrays of instances of parameterized types, the arguments to each element of the array may be different. The following code fragment demonstrates what would happen with the definition of MODEL **liquid_stream**.

```
MODEL liquid_stream(components : set OF symbol);

     pure[components] : pure_component;
     FOR i IN components CREATE
          pure[i] IS_A pure_component(i);
     END;

END liquid_stream;
```

The separation of declaration from construction preserves the name consistency and allows the creation of the desired object without the extra step of refinement. Each statement in the model can now be executed in the order that it was defined and the determination of whether the model is closed or not is trivial. If, at the closure of **test1**, constructors through the IS_A statement had not been provided, an error would have been flagged. In **test2** on line 17, an instance of *t1* is created using the parameterized type **test1**. If the values for the structural variables, *nstream* and *components* were not proved or were inconsistent with the formally declared arguments, then an error would also have been flagged.

### 3.5.2 STRUCTURAL PARAMETERS

Parameterized types, by themselves, add very little to the ASCEND III language. They formalize the construction of objects over structural parameters only and do not attempt to answer the important question of information hiding. However, by insisting on this formalism, the actual structural arguments may be bound to the instance. In the original language, at least two (2) passes would have been required to instantiate a model that was structurally parametric. There was no way to write a generic model and to be able to say in one construct how many parts it was to have. The arguments to the parametric type are limited to the types of **integer**, **symbol** and **set**.

Combined with the order of instantiation, parametric types allow *shallow* checking of an instance's state to make some operations more efficient. Though a detailed discussion of the semantics will not be done, the syntax of parameterized types will be described through some small examples.

```
MODEL test1(nstreams : integer;
            components : set OF symbol);

     stream[1..nstreams] : td_stream;
     stream[1] IS_A liquid_stream(components);
     stream[2..nstreams] IS_A vapor_stream(components);

END test1;

MODEL newtest(nstreams : integer;
             ntanks : integer)
             REFINES test1;

     tank[1..ntanks] IS_A square_tank;
     components := [‘C2H4’,’C3H6’];

END newtest;

MODEL newtest2 REFINES newtest;

     nstreams   := 4;
```

```
21     ntanks     := 12;
22
23 END newtest2;
```

EXAMPLE 13 Syntax of parameterized types

`MODEL` **test1** is the same as from Example 10. It is a model parameterized over structural variables with names *nstreams* and *components*. These names are part of the scope of the model. At the time of writing, the semantics imposed is that of argument passing by *value*. `MODEL` **newtest**, inherits from **test1** but commits to the value of the structural parameter *components*. This is done on line 14. It is then illegal to still have *components* in the argument list. Once a type is still parametric over some structural variables, *all* of the structural parameters must be present in the argument list of the `MODEL` declaration. From the standpoint of syntactic beauty this may be unattractive. In practice, the longest argument list seen had only five (5) elements (the entire distillation column library of the ASCEND III system, inclusive of the thermodynamics libraries were converted to accept parametric types). The insistence of this syntax is driven by the following:

> In using a model **m** from a library (which the modeler may not have developed), a modeler would have to look at the code for *all* models on the type hierarchy of **m** to determine what structural parameters would be necessary, in order to instantiate **m**. With the proposed syntax, only the code for **m** needs to be consulted.

A similar effect is observed when coding in ANSI C with prototypes as compared to coding in old style C.

In `MODEL` **newtest2**, all of the structural information has been specified, so that the parameter list is empty. Thus the argument list can grow and contract through the type hierarchy of a model. Finally, by being an argument list rather than a set, the ordering of the elements is important.

### 3.5.3 TYPES AS PARAMETERS

It is sometimes useful to be able to pass types as parameters. This enhances the expressiveness of the language for large or small scale modeling.

```
MODEL generic_stream(components : set OF symbol,
                    whattype : TYPE)

     s : molar_stream; (* say what base type *)

     SWITCH (whattype)
        CASE SRK_stream:
          s IS_A SRK_stream(components);
        CASE BWRS_stream:
          s IS_A BWRS_stream(components,2);
        CASE Ideal:
          s IS_A stream(components);
     END;

END generic_stream;
```

The new `TYPE` construct accomplishes this. `TYPE` is now a reserved keyword in ASCEND IV and may not be used as the name of a type.

### 3.5.4 INSTANCES AS PARAMETERS

The internal representation of ASCEND III is a directed acyclic graph or DAG. Because of the directed nature it is not possible to refer (through qualified naming) to an instance that is higher in the DAG. The root instance of the DAG is owned by the external environment and is called a *simulation*. It is possible to have an arbitrary number of simulations, but the language and environment does not explicitly allow the communication between simulations, as it would violate being able to access objects higher up on the tree. With parameterized types that allow instances as parameters and certain mild restrictions, it is then possible to refer to *external* instances. Among the reasons for the restrictions is the need to prevent the creation of cycles in the ASCEND III instance DAG. At the time of writing the restrictions are

- the instance that is eventually passed as a formal argument must be a simulation instance.
- the instance that is passed as a formal argument is *read only*.[2]The following example shows a model that accepts an instance of type **simple_tray** to be

2. It is easy for an instantiator to recognize the difference between a simulation instance and a structural instance of **integer**, **symbol** and **set**.

passed into a parameterized type.

```
1 MODEL column(tray_template : simple_tray;
2              ntrays : integer;
3              feedloc : integer);
4
5      tray[1..ntrays] : VLE_flash;
6      tray[1] IS_A
7           condenser(tray_template.state.components);
8
9      tray[2..(feedloc-1),(feedloc+1)..(ntrays-1)]
10                    CLONES tray_template;
11     ( ... )
12 END column;
```

EXAMPLE 14 Instance as Parameters

On line 7 of Example 14, the value of the *components* of the instance *tray_template* is used as the argument to construct an instance of a **condenser**. On line 10, the instance *tray_template* is copied explicitly to create the internal stages of the column model.

### 3.5.5 ALIASING REVISITED

ASCEND IV introduces the concept of an alias through the `IS` operator to overcome the overloading of the `ARE_THE_SAME` operator, which was discussed in Section 3.4.1.5. In effect the simplistic approach used by modelers to provide an interface to objects in ASCEND III has been formalized.

```
MODEL test3 REFINES test2;

     T : temperature;
     T IS t1.stream[1].T;

END test3;
```

EXAMPLE 15 Model **test3** of Example 12

In Example 15, `MODEL` **test3** also uses the new separation of type and construction features of ASCEND IV. The insistence of order, the deferral of instance construction and the use of the new aliasing construct, `IS`, allows the instantiator to save the step of constructing instance *T* then destroying it. The saving in instantiation time would be trivial in this case but would be significant in the following ASCEND III code listing:

```
reb IS_A Reboiler;      (* create the instance reb *)
c1 IS_A Column;         (* create the instance c1 *)
c1.tray[30],
     reb ARE_THE_SAME;  (* destroy instance reb *)
```

versus

```
reb : Reboiler;         (* state type of reb *)
c1 IS_A Column;         (* create the instance c1 *)
reb IS c1.tray[30];    (* use the IS alias construct *)
```

where instance *reb* is a fairly complex object.

## 3.6 REPEATED STRUCTURES

In earlier sections, the potential savings was shown that can be derived from the exploitation of repeated structures. The benefits are realized through *prototyping* and *cloning.* This will be discussed in the next few sections.

### 3.6.1 COARSE GRAINED STRUCTURES

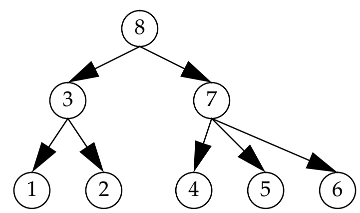
If it is recognized (manually or automatically) that many instances, all of the same type, are required, then the *coarse grained* structure of the type may be exploited. Only one of the *n* required instances would be subjected to a full semantic analysis yielding a *prototypical object*. This object may then be copied to provide the necessary objects of the same type, sharing structures wherever possible. The prototype and clone features have been implemented in ASCEND IV.alpha. It takes 3.0 seconds to instantiate a tray model with 5 chemical species (141 equations), but takes only 0.13 seconds to prototype it[3].

The prototype and clone metaphor require an ordering to instantiation and may

3. Times as measured in cpu seconds on a HP9000/715.

be applied recursively. The multipass lazy evaluation scheme of ASCEND III (which uses a queue) is thus replaced by a top down, depth first instantiator in ASCEND IV.

One of the open questions in the design of ASCEND IV is the level of responsibility that should be placed on the modeler versus that placed on the instantiator for making use of repeated structures.




Depth First - Bottom Up

FIGURE 4 An instance tree

Consider for example the instance tree shown in Figure 4. If the objects numbered 1 and 2, are **vapor_streams,** then a top-down, depth first instantiator would have seen them locally (see line 6 in Example 12) and could have made use of the repeated structures, to save time and memory. Assuming that the instances 4, 5 and 6 are also **vapor_streams**, then an instance of 4 could be created and copied to instances 5 and 6. However, if *all* the vapor_streams were *equivalent*, (using the definition as given earlier), then the instantiator would have missed an opportunity to save some work; it could have just created instance 1 and copied it to 2,4,5 and 6. In order to do this, it needs to have a global view of the final object that is to be created and to do some scheduling. A rough sketch of the modified instantiation algorithm would then be:

- Given a type description of the top level object, attempt to determine a type dependency graph, making note of the number of each instance type (explicit and implicit) and obtaining an estimate of the number of unique types that will occur in the instance.
- Using some metric determine which types should be used as prototypes when creating the instance tree. The objective function will be to minimize memory and to minimize instantiation time.

- Perform the real instantiation of the object, making use of the derived information.

The efficiency and complexity bounds of this algorithm need to be seen. The presence of implicit types complicates the analysis in the first step. To determine the number and frequency of *formal* types that would occur in the final object is straightforward enough; the same may not necessarily hold for implicit types.

It should be noted that modern C compilers will not attempt global optimizations. They do not optimize across function calls nor do they attempt optimizations across multiple files. These optimizations are simply too expensive. Programmers resort to *inlining* to achieve efficiency while maintaining code readability [Dow93].

If it is assumed that a global analysis is prohibitively expensive and a modeler through provided language constructs is able to instruct the instantiator, then great efficiencies can be achieved. For example in a model of a large chemical process plant, there may be 500 to 1500 streams. If it assumed that there are really only four (4) unique stream types, i.e., liquid and vapor streams defined over two different sets of chemical species, then a modeler at the start of large problem could do the following:

```
% l1 IS_A_PROTOYPE liquid_stream(stream_set1);
% v1 IS_A_PROTOYPE vapor_stream(stream_set1);
% l2 IS_A_PROTOYPE liquid_stream(stream_set2);
% v2 IS_A_PROTOYPE vapor_stream(stream_set2);
```

Here the new language construct `IS_A_PROTOYPE` is used to create prototype instances of the different kinds of streams. If these instances are then stored in a global prototype library using the names of their types as the primary key and an encrypted version of their arguments used as a secondary key, then the determination of whether a repeated structure exists which may be exploited is very fast. An instantiation algorithm which is applicable at any stage of instantiation is then:

- When a new model scope is entered, perform a local analysis of the instance types that need to be created. For each type check the prototype library for compatible prototypical objects. For those that exist, clone the object from the library. For those that do not, perform a *local* analysis to see if *n* objects of the same type are needed.
- If *n* > *1*, create a prototypical object, and clone it *n-1* times.
- If *n* = *1*, create the object.

It should be pointed out that the instantiator *cannot* add any structures created locally to the global prototype library as it may add an object of implicit type. The reason for this is as follows:

- The instantiator decides to add an object *o* of type **t** to the global prototype library.
- To determine *quickly* if an object of type **t** is already in the prototype library, it uses a shallow comparison (using the type **t** and the arguments of the object *o*), to perform the look up. A deep comparison could be too expensive. Assuming that an object of type **t** did not already exist, it would be successfully added to the prototype library.

If the object *o* happened to be of implicit type **t'** rather than the explicit type **t**, the prototype library would have become corrupted. A subsequent shallow query of the library for the type **t** would succeed, with fatal results.

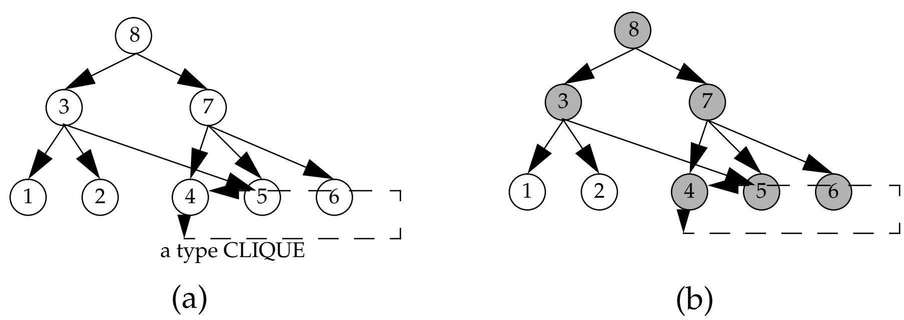



FIGURE 5 Instance DAG with a type CLIQUE

In order to avoid this situation, the instantiator could keep track of the true type of all objects and to update the type information whenever operations are performed that can create implicit types.

This updating may be expensive. If an operation is performed on instance 4 in Figure 5a which creates an implicit type for that object, then *all* objects on *all* paths back to the root instance (i.e., 4, 7 and 8) now possess implicit types. In addition the type propagation mechanism of `ARE_ALIKE`, which in this example places objects 4, 5 and 6 in the same clique, would require updating the information for all objects on all paths leading from 5 and 6 back to the root. Thus object 3 would have been sullied as well (see Figure 5b). It is thought that maintaining this implicit/formal type information is too expensive.

The `IS_A_PROTOYPE` construct provides a reasonable way of achieving fast instantiation, without resorting to the significantly more onerous option: making implicit types illegal. In this way ASCEND IV behaves more like a prototype based object-oriented system, while offering the efficiency that arises from class based systems.

### 3.6.2 FINE GRAINED STRUCTURES

In the previous section, the ability was shown to exploit repeated coarse grained structures. Significant potential also exists for exploiting repeated structures at a finer level of granularity. This happens with the explicit iterators used for constructing groups of relations. In ASCEND III a group of similar relations may be constructed by use of the `FOR` construct. This is shown in Example 16.

In this example the linear system $\mathbf{Ax = b}$, is formed, and there will be $m$ equations all the form $\sum_j A_{ij} x_j = b_i$. Structurally all of the equations are the same, i.e. they have the same variable incidence pattern. This is not always the case, as some relations may involve set operators which can change the number of terms in the relation. However these deviations are easy to recognize and involve a single scan of the static expression structure that represents the relation. For a group of relations which are structurally the same, an instantiator can perform the loop index evaluation and create the necessary relation instances. It can then perform

```
MODEL matrix_multiply(m : integer;
                     n : integer);

     A[1..m][1..n] IS_A solver_var;
     x[1..n], b[1..n] IS_A solver_var;

     FOR i IN [1..m] CREATE
        eqn[i]: SUM(A[i][j] * x[j] | j IN [1..n]) = b[i];
     END;

END matrix_mulitply;
```

EXAMPLE 16 Fine Grained Repeated Structure

semantic analysis on a *single* relation. The variable incidence pattern derived whilst doing the semantic analysis can then be applied to all the relation instances. Only one check on the correctness of the relations has to be done, and only one copy of the relation structure needs to be stored. It is interesting to note that the above model could have been written in the following way:

```
MODEL dot_product(n : integer)

     row[1..n] IS_A solver_var;
     x[1..n], b IS_A solver_var;

     dproduct : SUM(row[i] * x[i] | i IN [1..n]) = b;

END dot_product;

MODEL matrix_multiply(m : integer;
                     n : integer);

     A[1..m] IS_A dot_product(n);
     x[1..n] IS_A solver_var;
     b[1..m] : solver_var;

     x, A[1..m].x ARE_THE_SAME; (* a single x vector *)
     FOR i IN [1..m] CREATE
          b[i] IS A[i].b;
     END;

```

```
22 END matrix_multiply;
```

In this case the instantiator using the coarse grained prototyping algorithm would see the potential for repeated structure on line 13, and the result would have been the same, in terms of speed and memory requirements. This proves the validity of the fine grained analysis.

### 3.6.3 SHALLOW OPERATIONS

The use of implicit types requires that operations such `ARE_THE_SAME` use deep comparisons, rather than shallow comparison. In deep comparisons two instances that are to be merged have to checked recursively for compatibility. For a shallow comparison, just the signatures need to be checked. If compatible, then one object is deleted and the references adjusted. If there is a cheap way to keep track of implicit types (perhaps through dirty and clean bits), then an instantiator would be able to tell when a deep versus a shallow operation should be done. The efficiency of maintaining this information remains to be seen.

## 3.7 MEMORY

It is expected that the cost to represent a problem in a high level modeling language will be higher than that of a low level language. This is *reasonable*, given the wider range of queries to which a high level modeling language/environment can respond. However if this cost becomes excessively high, then, regardless of their potential utility, high level modeling languages will fail. These issues were discussed at length in Chapter 2. The memory management issues of ASCEND IV are discussed in the next few sections.

### 3.7.1 VARIABLES

ASCEND III uses the *everything is an object* philosophy in its implementation. This philosophy also applies to the children of atoms (sub atomic parts). This is done in a space efficient manner, but, being full instances, the cost is greater than the cost of representing the equivalent double precision number. The ASCEND III system also employs a very loose definition of what constitutes a *variable* and leaves the determination of the minimum state that a variable must possess to the solver being invoked. The definition for a compiler variable is any atomic instance which is rooted at the **real** type hierarchy, i.e., any REAL_ATOM_INSTANCE. All of the solvers developed and attached to the ASCEND III system, however, employ a much more restricted definition; a solver variable is any atomic instance which is rooted at the **solver_var** type hierarchy. (see Figure 7 in the Appendix). The definition of a **solver_var** is dynamic and is determined by what base libraries are loaded when starting an ASCEND III session. For the example PPP the cost a solver variable was 200 bytes or a 168 byte overhead.

By raising the minimal definition of a variable to that of the current **solver_var**, the cost of a variable can be reduced to 96 bytes. Projected saving in memory is 9 MB at 100,000 variables. This requires the introduction of a new fundamental kind of instance, a SOLVER_REAL_INST, to augment the REAL_ATOM_INST. The definition is

```
struct SolverRealInstance {
     [...] object overhead;
  double lower, upper, nominal, value;
  boolean fixed;
  boolean integral;
  Dimensionality *dims;
};
```

These are purely internal changes that do not affect the language in any other way, save for the implementation of code that makes queries on variables.

### 3.7.2 CONSTANTS

Most languages have the notion of a **constant**. A constant is a parameter that has its value set at creation, and this value is immutable. A compiler can treat a constant in special ways because of this immutability. It may be constant folded and put into reserved storage locations. In ASCEND III there are no **constant**s. The constant type shown in the type library of the example PPP is just an artifact of the modeler’s naming convention (but in this case does have the same semantics). There is a non-trivial number of them. With over 33,000 instances of constants or *twice* as many variables as the solver thinks are in this problem, at a cost of 64 bytes apiece (an excess of 56 bytes), it may be worthwhile to introduce this concept into the language. Projected saving in memory is 11.0 MB at 100,000 variables. The impact on speed of solution could also be nontrivial.

### 3.7.3 RELATIONS

All of the issues discussed with respect to the use of repeated structures drastically affect the memory required for the representation of relations, by simply reducing the number of relation structures that have to be maintained. For the PPP example these schemes have resulted in a 37 MB savings by exploiting the coarse grained repeated structures. The other major savings will come from a fundamental change in the internal representation of relations. This is discussed in detail in the next chapter.

## 3.8 OTHER

### 3.8.1 VERY LARGE SCALE MODELING

In the earlier sections, the *prototype and clone* metaphor was proposed as a means of improving instantiation speed and minimizing memory consumption. However, as model development moves towards the root of the instance DAG, models start to become more specialized, and the ability to make use of repeated coarse grained repeated structures diminishes and eventually stops.

This is in keeping with reality. It is possible to buy a generic heat exchanger for water service off the shelf. Disposal of these units at decommissioning is also fairly easy. It is perhaps less easy to acquire and dispose of a 100 tonne/hour Bayer liquor evaporator. A *one-a-of-a-kind* object is required [US91]. A simple application of a prototype and clone metaphor starts becoming *more* expensive in memory. In addition these difficulties arise where instantiation speed is needed most; i.e., when the model becomes large. The solution to this dilemma is the provision of *reference* semantics.

A large model will not be fully developed and debugged in a single modeling session. In order to propagate efforts from one modeling session to the other, some form of *persistent representation* is required. These issues were discussed in some detail in Section 2.6.3. Given that a persistent representation of a model exists, then a new modeling session would first involve restoring this model from its database. If the example PPP is used and the instances *c3_c4* and *c3_splitter* had been previously stored, then the following code issued from a command line will restore these instances. The `IS_A` construct is being overloaded for convenience.

```
1 % c3_c4 IS_A Column30;
2 % c3_splitter IS_A Column164;
```

Using the instance as a parameter feature as discussed in Section 3.5.4, it is possible to pass these external instances into a model definition as is done on line 16 below.

```
3 % %%
4 > MODEL coldsection(c1 : Column30;
5                     c2 : Column164);
6
7      m1 : mixer;
8      FOR i IN c1.components CREATE
9           connect[i]:
10          c1.reboiler.f[i] = c2.feed.f[i];
11     END;
12     ( ... )
13  END coldsection;
14 > %%
15 %
16 % qqq IS_A coldsection(c3_c4,c3_splitter);
17 %
```

Instances *c3_c4* and *c3_splitter* remain unchanged; they are simply referred to, *not* copied. If an error occurs whilst constructing the instance *qqq* of **coldsection**, the instance may be deleted without affecting the instances *c3_c4* and *c3_splitter* and reconstructed with the only cost being that of reinstantiating the statements introduced in this model. If, however, restrictions were not placed on instances passed as parameters, it would be possible to corrupt these instances, and they would be have to be deleted at the same time as *qqq*.

With these capabilities the cost of building arbitrarily complex models rapidly and with efficiency of memory use is possible through all phases of the modeling exercise. This has been implemented. The cost of reinstantiation for `MODEL` **coldsection** is less than 1 second.

### 3.8.2 FUNCTION AND GRADIENT EVALUATION

In Chapter 2, it was argued that judicious use of code generation could yield significant improvements in function and gradient evaluation times. The use of both coarse and fine grained repeated structures may be used to keep the generated code small and yield reasonable compilation times for the generated code. Using the *glass-box* code generation features implemented in ASCEND IV.alpha, a 6.5 and 5.9 time speed up for function and gradient evaluation respectively, over pure interpreted code was achieved. These results were obtained with about 90% compiled code and 10% interpreted code. The protocol to allow compiled external code to reside seamlessly with interpreted code in a manner transparent to the user was made possible through the new PATCH construct.

The details of the code generation mechanism and the PATCH are discussed in Chapter 4.

### 3.8.3 DELETION

Destroying a part of a highly connected structure, such as an instance DAG, can be a slow process; an orderly shut down is required. With a fewer number of structures present and with purely reference semantics, deletion of portions of a simulation becomes cheaper.

### 3.8.4 `ARE_ALIKE`

Type propagation with the `ARE_ALIKE` construct breaks just about every feature discussed in this chapter. In practice this language feature of ASCEND III has been rarely invoked. In its implementation `ARE_ALIKE` involves some $O(n^2)$ algorithms. `ARE_ALIKE` is not supported in ASCEND IV.alpha.

## 3.9 DISCUSSION

In this chapter, a new language ASCEND IV has been proposed which addresses some of the weaknesses of its predecessor ASCEND III. It maintains and augments the expressiveness of the ASCEND IIIl language, whilst being more efficient in the areas identified in Chapter 2 as being essential for a Very Large Scale Modeling environment. This has been demonstrated by using full and experimental implementations of many of the features described. The current and projected results for the example PPP are shown in Table 4.

**Table 4: Instantiation Statistics**

| | ASCEND III | ASCEND IV[1] alpha | ASCEND IV[2] |
|---|---|---|---|
| # Relations | 17516 | 17516 | 17516 |
| # Variables | 18113 | 18113 | 18113 |
| Size (MB) | 65.36 | 27.66 | < 20.0 |
| # Instances | 247380 | 247380 | <125000[3] |
| # Formal Types | 43 | 43 | > 43 |
| # Arrays | 13581 | 13581 | 13581 |
| Instantiation Time (sec) | 985.76 | 25.0 | < 10.0 |
| Function Time (sec) | 13.19 | 2.02 | < 1.0 |
| Gradient Time (sec) | 28.73 | 4.90 | < 2.0 |

1. data obtained using prototype implementations of the new language.
2. projected for full implementation of ASCEND IV.
3. reduction in the number of instances because of the new SOLVER_REAL instance and introduction of constants.

Compared with ASCEND III, the savings in instantiation time for ASCEND IV.alpha, are noteworthy, the savings in memory satisfactory while the savings in evaluation time are somewhat disappointing, given the results seen in Section 2.6.4.

The largest problem on which ASCEND IV.alpha has been tested is an 86,000 variable, 79,000 relations model of an ethylene plant, and it requires 175 MB for problem representation, 15.0 and 55.0 seconds for function and gradient evaluation respectively and takes 740.0 seconds to instantiate. These numbers are expected to be lower by a factor of 2 to 3, when ASCEND IV is fully implemented, as there are known inefficiencies in the prototype implementations.

Further implementation details and results, particular with respect to relation representation will be presented in Chapter 4. It is thought that ASCEND IV can be almost as efficient in terms of representation, instantiation speed and evaluation speed as a lower level language.

Perhaps the primary failure of ASCEND IV is that it does not address the difficult issue of information hiding or information filtering. It is also not minimal in its use of constructs. The detailed semantics of some operations and the efficiency with which they can be implemented remains to be seen. In addition, ASCEND IV also requires somewhat greater modeler responsibility; this may or may not be a failing.

# 3.10 APPENDIX

## 3.10.1 MODEL FLOWSHEET

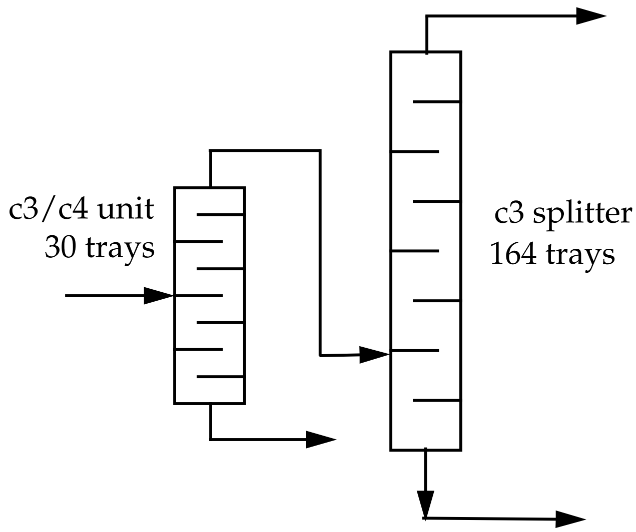



FIGURE 6 C3 separation unit

## 3.10.2 SYNTAX

A simplified description of the language ASCEND IV is presented below.

### 3.10.2.1 FUNDAMENTAL TYPES

real

boolean

symbol_index (now symbol)

integer_index (now integer)

integer (now mut_integer)

symbol (now mut_symbol)

set

TYPE

### 3.10.2.2 MODELING CONSTRUCTS

IS_A

IS

IS_REFINED_TO
IS_A_PROTOTYPE
:
ARE_THE_SAME
ARE_ALIKE (dropped)
CLONES
ATOM
MODEL

#### 3.10.2.3 GRAMMAR

```
model           : MODEL model_name;
                | MODEL model_name REFINES refined_type;
                | MODEL model_name ( formal_args );
                | MODEL model_name ( formal_args )
                  REFINES refined_type ;
                ;

formal_args     : formal_arg
                | formal_args ; formal_arg
                ;
formal_arg      : arg : type_of_arg
                ;
arg             : identifier extension
                | arg extension
                ;
extension       : [ ]
                ;
type_of_arg     : integer_index
                | symbol_index
                | TYPE
                | set OF integer_index
                | set OF symbol_index
```

```
;
```

### 3.10.3 INSTANCE COUNTS BY TYPE FOR MODEL PPP

```
Pitzer_component                                  588
Pitzer_mixture                                    196
Rackett_component                                 591
UNIFAC_constants                                  197
UNIFAC_mixture                                    197
UNIFAC_parameter                                  1182
UNNAMED ARRAY NODE                                13581
boolean                                           35823
column164                                         1
column30_eq                                       1
constant                                          33393
energy_rate                                       589
equilibrium_mixture                               196
factor                                            2952
fraction                                          792
gas_constant                                      1768
integer                                           35633
liquid_stream                                     197
mixture                                           198
molar_energy                                      5306
molar_rate                                        2376
molar_stream                                      198
molar_volume                                      2356
mole_fraction                                     1973
partial_component                                 1179
pressure                                          788
propadiene                                        1
propane                                           1
propylene                                         1
real                                              55327
recovery_backend                                  1
relation                                          17516
relative_volatility                               784
scaling_constant                                  3932
set                                               4534
symbol                                            4927
td_condenser_PPP                                  2
td_reboiler_PPP                                   2
```

| | |
|---|---|
| td_simple_feed_tray_PPP | 2 |
| td_simple_tray_PPP | 188 |
| temperature | 197 |
| vapor_liquid_stream | 2 |
| vapor_stream | 196 |

### 3.10.4 GENERAL INSTANCE TREE NUMBERS FOR MODEL PPP

```
Number of models and complex atoms: 73170
Number of atom children instances: 125597
Number of relations: 17516
Number of array instances: 13581
```

## 3.10.5 A TYPE HIERARCHY

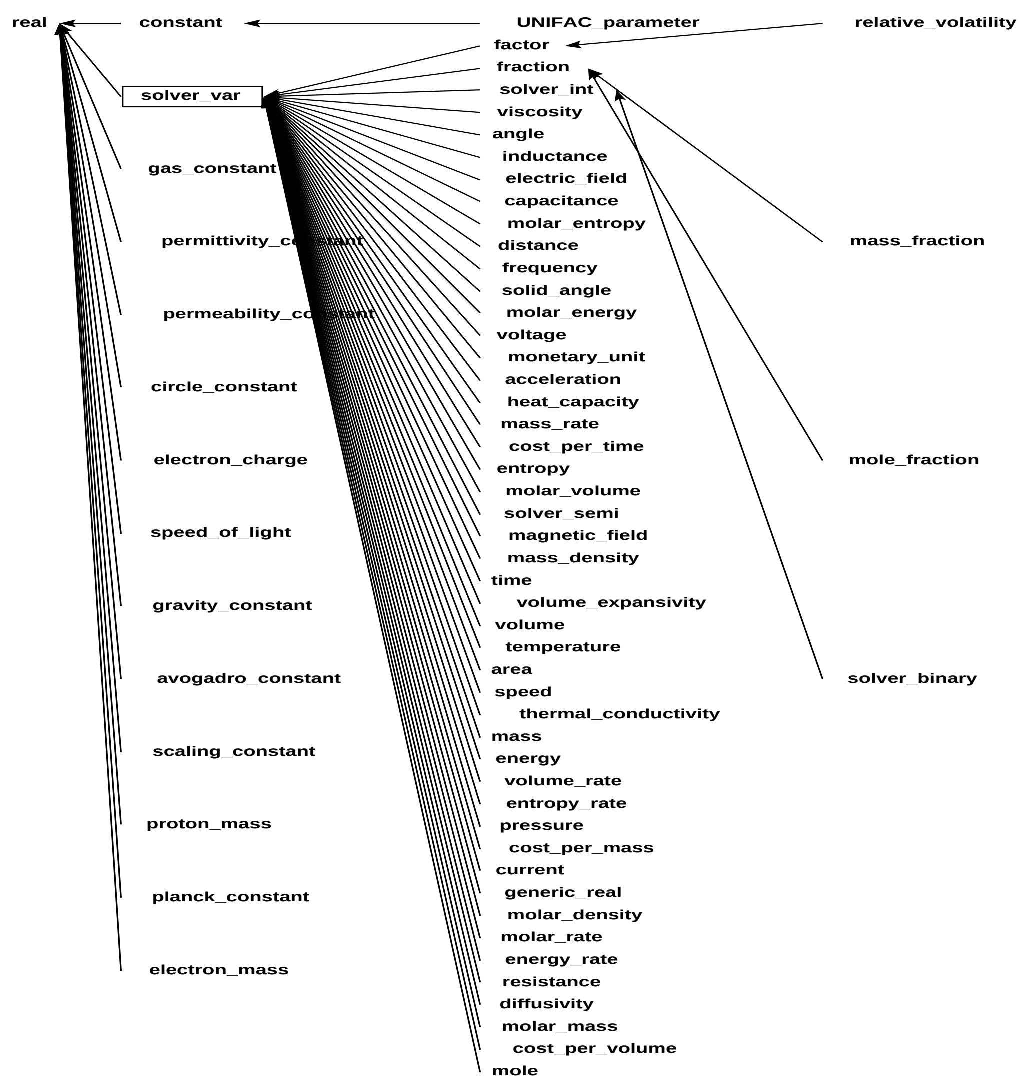



FIGURE 7 Type hierarchy for atoms

# 3.11 REFERENCES


[Bar92] P. I. Barton. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, 1992.

[CU91] C. Chambers and D. Ungar. Iterative type analysis and extend message splitting: Optimizing dynamically-typed object-oriented programs. *Lisp and Symbolic Computation: An International Journal*, 4(3), 1991.

[Dow93] K. Dowd. *High Performance Computing*. ORLY, 1 edition, June 1993.

[EN94] J. Eborn and B. Nilsson. Object-oriented modelling and simulation of a power plant. application study in the K2 project. Technical Report ISRN LUTFD2/TFRT–7527–SE, Department of Automatic Control, Lund Institute of Technology, December 1994 1994.

[Epp89] T. G. Epperly. Implementation of an ASCEND interpreter. Technical report, Engineering Design Research Center, Carnegie Mellon University, 1989.

[Mey88] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[Nil93] B. Nilsson. *Object-Oriented Modeling of Chemical Processes*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[Ost94] J. K. Osterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PEWW91] P. Piela, T. Epperly, K. Westerberg, and A. Westerberg. An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering*, 15(1):53–72, 1991.

[Pie89] P. Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennyslvania, April 1989.

[PMW93] P. Piela, R. McKelvey, and A. Westerberg. An introduction to the ASCEND modeling system: its language and interactive environment. *J. Manage. Inf. Syst.*, 9(3):91–121, Winter 1992-1993.

[US91] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, 4(3), 1991.

# CHAPTER 4 NEW LANGUAGE FEATURES

## 4.1 ABSTRACT

This chapter looks at some of the language features introduced into the ASCEND III language to allow it to be useful as a language in a VLSM environment. Emphasis has been placed on features that will improve the speed of building, efficiency of representation and speed of solution of arbitrarily large and complex mathematical models.

## 4.2 INTRODUCTION

In Chapter 3, a detailed proposal was presented for a new language to support a VLSM environment. That discussion was supported by results that were obtained from doing prototypes and full implementations of a number of new language features. The resulting product is called ASCEND IV.alpha which is an intermediate environment which, when completed, will be ASCEND IV. Some of the implementation details are given in the following sections.

## 4.3 RELATIONS

Relations are perhaps the most expensive items to process and to represent in a modeling language. For large sparse systems that will be solved by some numerical technique, a relation has to be able to respond to some minimal queries. These include:

- to return the sparsity pattern of its incident variables so as to allow construction of the sparse matrix representation of the problem.
- to return the residual $f(x)$ for given a vector $x = \hat{x}$.
- to return the derivative vector $\frac{\partial}{\partial x}f(x)$ at $x = \hat{x}$, as well as possibly higher order derivatives.

A modeling environment needs to provide a solver with this information efficiently. In addition symbolic processing and analysis of a relation is sometimes necessary. A modeling language also has to support relations written in a language other than the provided modeling language. Two types of *external relations* have been identified, black-box and glass-box external relations.

The ability to support these different requirements efficiently has led to the development of four (4) different data structures and their support routines in the ASCEND IV.alpha environment. These are

- token relations.
- opcode relations.
- glass-box relations.
- black-box relations.

Over the life cycle of a large scale modeling problem it is expected that all of these relations will be used in different proportions by the ASCEND IV environment. In the early stages of model development where speed of solution is not paramount and symbolic analysis of relations is required, 90 to 100% of the relations used will be token or opcode relations. For mature models, it is expected that 10% of the relations will be opcode relations, 85% will be external glass-box relations, with the rest being external black-box relations.

Protocols in the ASCEND IV system have been set up to accommodate the different relation needs over the entire spectrum of model development and to do so automatically. The purpose of each of these types of relations will become clear in the following sections.

### 4.3.1 ASCEND III RELATIONS

In the ASCEND III system, relation instances hold pointers to relation structures (see Figure 8). These relation structures in turn hold

- a variable list of pointers to the REAL_ATOM_INSTances that are incident upon the relation.
- a list of tokens associated with the left hand side and the right hand side of the relation, held in postfix.
- space for the residual, Lagrange multiplier, nominal, dimensionality, and the type of the relation. The type tells whether the relation is an inequality, equality or an objective function.

The token list points to relation terms[1], where each term is a union of the different types of terms that can exist in a relation. These include:

- real tokens (numeric constants and their dimensionality).
- integer tokens.
- variable tokens, which hold an index into the variable list of the given variable.
- function tokens, which includes all the standard transcendental functions and some common polynomials.
- unary operators (unary minus and unary plus).
- binary operators (plus, minus, divide, times and power).

The inclusion of the unary and binary tokens allows a relation to be *wired up* as a threaded tree, which allows infix scanning at no additional cost as the size of the union is dominated by the cost of representing real tokens.

1. relation tokens and relation terms are synonymous.

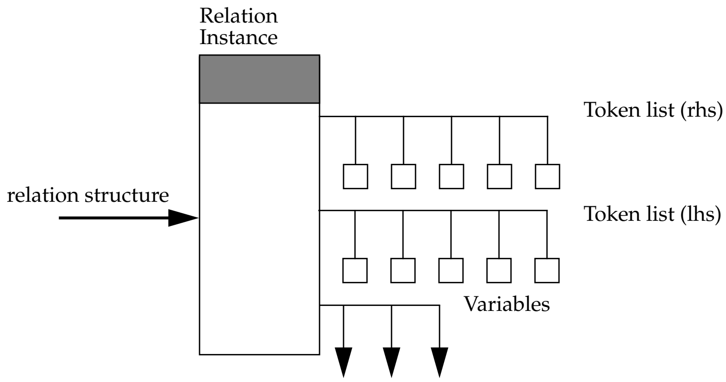



FIGURE 8 Original Relation Structure

### 4.3.2 TOKEN RELATIONS

The representation of relations in ASCEND III is convenient but expensive. Each relation term costs 24 bytes (see the Appendix). More importantly, as each relation instance *owns* the relation structure which it is associated with, no mechanism exists to *share* relation structures among multiple relation instances.

In order to allow the sharing of relations, a level of indirection is introduced into the original representation of Figure 8 to yield the representation shown in Figure 9. In addition, a reference count for destroying the relations is associated with each relation structure. The relation structure is actually a union of the four (4) types of relation structures identified in Section 4.3.

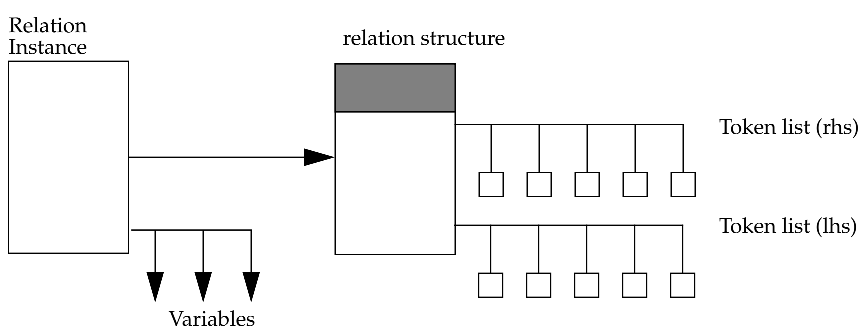



FIGURE 9 New Token Relations

In addition, the variable list is now associated with the relation instance rather than with the relation structure. Although expensive, the ASCEND III token relation data structure is useful for symbolic analysis inclusive of symbolic differentiation, as well being a useful intermediate data structure when performing instantiation of relations. Token relations are hence supported in ASCEND IV.

### 4.3.3 OPCODE RELATIONS

As discussed in Chapter 2, the opcode representation is a very efficient representation for relations in terms of both memory and evaluation speed (when using an interpreter to perform evaluations). Though there are many possible representations for opcodes, a simple but effective model is to use

- a vector of double precision numbers to represent the numeric constants.
- an integer vector of opcodes, to represent the algebraic operators and offsets into a variable list and the constant vector.

This representation is shown in Figure 10. The cost of this representation is approximately 25% that of the cost of token relations. However, symbolic analysis (such as dimensionality checking of a relation) when using this representation is clumsy. Opcodes will have to be expanded into token relations to perform these analyses, or these analyses should be done prior to mapping into the opcode representation.

Although the data structures have been set up to accommodate opcode relations, the interpreter to perform function and gradient evaluations over these structures is not fully implemented in ASCEND IV.alpha.

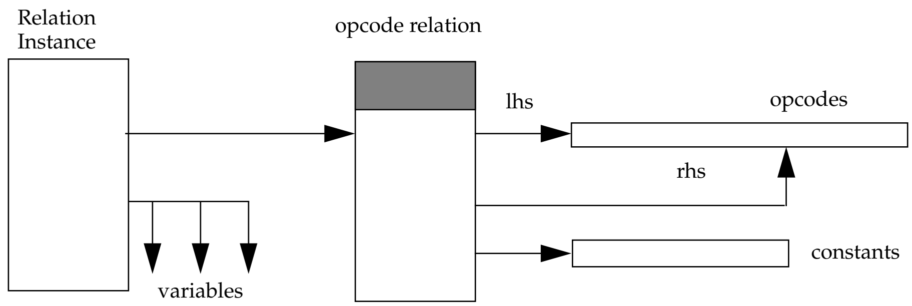



FIGURE 10 Opcode Relations

### 4.3.4 GLASS-BOX RELATIONS

External glass-box relations are expected to be the workhorse relations of ASCEND IV. An external glass-box relation is a relation that can answer the minimal set of queries identified in Section 4.3 and hides none of its sparsity information from the main environment *but* uses external code to perform its evaluations. To the rest of the system, an external glass-box relation behaves *exactly* like a token relation. The internal representation is shown in Figure 11.

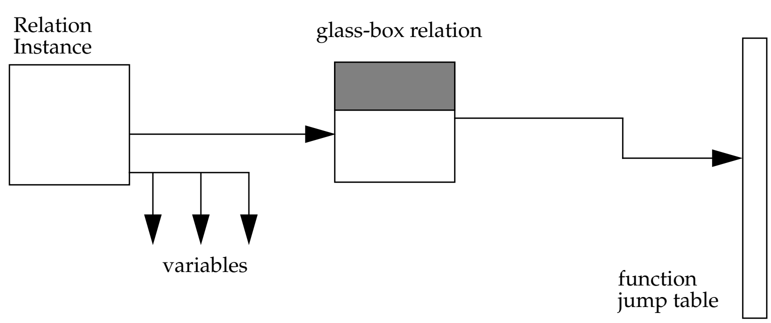



FIGURE 11 Glass-box Relations

The workings of an external glass-box relation are best shown through an example. In the following code fragment:

```
a_relation:  fast_flash(x[1],x[25],y[12]; 2);          1
```

*a_relation* is the name of the external glass-box relation. **fast_flash** is a symbolic reference to the code which does the function and gradient evaluations. The comma delimited terms in parentheses are the names of ASCEND IV variables that are incident upon the relation, in this case variables *x[1]*, *x[25]* and *y[12]*. The integer 2, says that the code to do the evaluations is the 2nd function in the block of external code for **fast_flash**. With this syntax, all the information necessary to construct a sparse matrix and to perform evaluations is available. The flexibility to fix and free variables is still maintained.

The syntax in extended *BNF* is shown in Figure 12. The significance of the **optional_scope** qualifier seen in the grammar will be explained later.

```
glassbox_relation : name : IDENTIFIER ( varlist ; INTEGER )
                    optional_scope

name : INDENTIFIER ([ IDENTIFIER ])*

qualified_name : name (. name)*

varlist : qualified_name (, qualified_name)*

optional_scope : qualfied_name
```

FIGURE 12 Syntax for Glass-box Relations

To make the ASCEND IV system aware of the external relations, the protocol shown in Figure 13 has been set up for *registering* the code. These external functions may be statically or dynamically linked to the ASCEND IV binary. For dynamically linked code, the IMPORT statement of ASCEND III was extended to deal with libraries of compiled code. For example, the following ASCEND IV code fragment will link a dynamically loadable object,

```
IMPORT  reactor_1_Init  FROM  libreactor;
```

**libreactor.so**[2], locate the routine **reactor_1_Init** and execute it. This routine may contain arbitrary statements, but, if it includes the registration protocol described in Figure 13, then the ASCEND IV system will have access to all the external function and gradient evaluation code.

```
typedef int
ExtEvalFunc(int *mode, int *m, int *n,
           double *x, double *u, double *f, double *g);

extern int
CreateUserFunction(CONST char *name,    /* a symbolic handle */
                  ExtEvalFunc *init,    /* initialization code */
                  ExtEvalFunc **value,  /* residual function table */
                  ExtEvalFunc **deriv,  /* 1st derivatives func table */
                  ExtEvalFunc **deriv2, /* 2nd derivatives func table */
                  char *help);          /* help information */


/*
 * the function and derivative tables are arrays of pointers to functions.
 * all arguments to the evaluation routines are by reference to support
 * mixed language calls.
 */
```

FIGURE 13 External Glass-Box Relation Registration

### 4.3.5 BLACK-BOX RELATIONS

Black-box relations are a set of relations which are of the form

$$y = f(x, u) \tag{17}$$

where $y \in \Re^m$ represent outputs, $x \in \Re^n$ are the inputs to a system of equations and $u$ represents parameters to the problem. Internal to a black-box there will usually be a set of equations

$$g(y, x, u, z) = 0 \tag{18}$$

used to solve for variables $z$, which are strictly internal to it. The black-box will have to solve these equations, in order to compute the residual vector given as

2. The extension .so is automatically determined based on the host operating system. HPUX uses .sl while OSF, and SUNOS use .so.

$$y - f(x, u) = 0 \qquad (19)$$

and the *mxn* sensitivity matrix

$$\frac{\partial y}{\partial x} \qquad (20)$$

This information is needed by most modern algorithms for solving the larger system of equations in which these black-box equations are embedded. Arguably this is not the most efficient way to solve the overall system of equations, as it involves repeated convergence of the inner set of equations represented by the black-box. Westerberg [WHMW79] gives convincing arguments why this is so. However, it may be necessary to deal with external codes that have a limited interface and/or legacy or proprietary codes. Black-boxes are accommodated in ASCEND IV through a syntax similar to that used for glass-box relations.

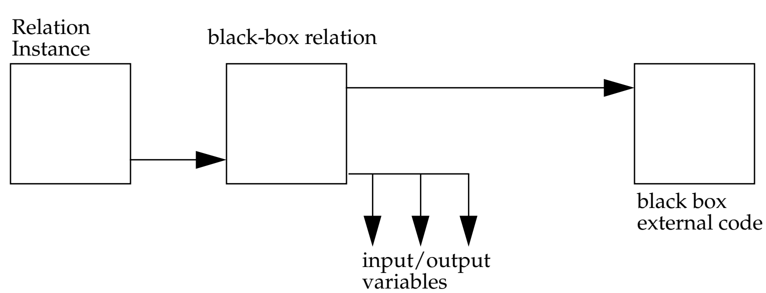



FIGURE 14 Black-Box Relations

The syntax is shown in Figure 15. With this syntax the ASCEND IV compiler has enough information to check the validity of the input and output arguments which must resolve to REAL atomic instances. The DATA argument allows additional information to be passed to the external routine, where this information may be anything that is accessed through an ASCEND IV model. The instantiator verifies the existence of the external code through the symbolic handle and then expands the **blackbox_relation** statement into *m* black-box relations of the form

$$y_i - f(x, u) = 0 \tag{21}$$

where each relation has *all* of the incident input variables but only one (1) output variable, thus creating the dense sparsity pattern of Figure 16 (see also Figure 18).

```
blackbox_relation : name : IDENTIFIER ( input_args ;
                                        output_args
                                        (; data_args)* ) ;

name : INDENTIFIER ([ IDENTIFIER ])*

qualified_name : name (. name)*

input_args : varlist : INPUT

output_args : varlist : OUTPUT

data_args : name : DATA
```

FIGURE 15 Syntax for Black-box Relations

Lastly each expanded relation is made to point to a data structure that maintains the information necessary to evaluate the residuals and gradients.

Black boxes tend to calculate *all* their outputs for a given set of input variables and parameters. In addition the computation of these outputs and the associated gradient matrix may be expensive (the solution process could be a complex simulation for example). A solver however, might not require this information in any sequence as each expanded relation is not treated specially. To ensure efficiency of computation of these residuals and gradients at a consistent point by a solver, a unique *node stamp* is associated with each **blackbox_relation** statement. By checking the node stamp and the vector of input variables and parameters, the black-box routine can determine whether to redo its calculations or simply to return the appropriate elements of its solution vector or rows of its sensitivity matrix.

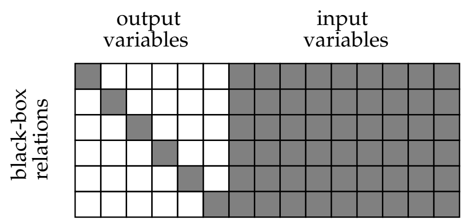



FIGURE 16 Sparsity Pattern of Black-Box Relations when expanded

The process of making the ASCEND IV system aware of these black-box relations is very similar to that used for glass-box relations but requires some additional input information concerning the number of input and output variables and a routine for initialization. The modified registration code is shown below.

```
extern int
CreateUserFunction(CONST char *name,
                   ExtEvalFunc *init,
                   ExtEvalFunc **value,
                   ExtEvalFunc **deriv,
                   ExtEvalFunc **deriv2,
                   unsigned long n_inputs,
                   unsigned long n_outputs,
                   char *help);
```

FIGURE 17 Black-box relation registration protocol

## 4.4 PROCEDURES

As was previously discussed, the ASCEND language description allows `PROCEDURES` to be bound to instances. These `PROCEDURES` may introduce no new state to an instance but can be used for different configurations of state. For example they are used extensively in setting degrees of freedom, scaling, giving initial guesses and for setting values of parameters for an instance. The ASCEND

IV inheritance model is that of textual inclusion *except* for `PROCEDURE`s. `PROCEDURE`s may be overwritten, with new procedures replacing any old procedures of the same name.

In ASCEND III, in order to maintain a consistency of naming, while maintaining the functionality of the old procedure, the statements in the old procedure had to be manually copied. The introduction of the *type access* of procedures in ASCEND IV, which is loosely based on the class access operators in C++, the **::** syntax allows reference to procedures by any type on the inheritance hierarchy of a model. This is shown in the Example 22b. The result has been much cleaner and condensed model descriptions.

```
MODEL test1;

   x[1,2] IS_A generic_real;

   INITIALIZATION
   PROCEDURE clear;
      x[1,2].fixed := FALSE;
   END clear;
   PROCEDURE specify;
      x[1,2].fixed := TRUE;
   END specify;

END test1;

MODEL test2 REFINES test1;

   z IS_A generic_real;

   x[1] + x[2]*z1 = 20.9;

   INITIALIZATION
   PROCEDURE clear;
      x[1,2].fixed := FALSE;
      z.fixed := FALSE;
   END clear;

END test2;
```

(a)

```
MODEL test1;

   x[1,2] IS_A generic_real;

   INITIALIZATION
   PROCEDURE clear;
      x[1,2].fixed := FALSE;
   END clear;
   PROCEDURE specify;
      x[1,2].fixed := TRUE;
   END specify;

END test1;

MODEL test2 REFINES test1;

   z IS_A generic_real;

   x[1] + x[2]*z1 = 20.9;

   INITIALIZATION
   PROCEDURE clear;
      RUN test1::clear;
      z.fixed := FALSE;
   END clear;

END test2;
```

(b)

EXAMPLE 22Type Access for Procedures

Consistent with the philosophy of an open environment, external routines may be accessed from within an ASCEND IV procedure. The syntax is:

```
external_procedure : EXTERNAL IDENTIFIER ( list ) ;

list : name (, name)*
     | SELF ,list

name : INDENTIFIER ([ IDENTIFIER ])*
```

The process of registration of external procedures is very similar to that for external black-box and glass-box relations. Very little type checking is performed. The semi-reserved SELF keyword is used to describe the object in which the PROCEDURE definition was made. These external procedures have complete access to the ASCEND data structures.

## 4.5 CODE GENERATION

In order to achieve high efficiency of function and gradient evaluation, a code generation model was adopted: generate code in an intermediate language, compile to machine code and then call that code to do all necessary evaluations. C was the target language used in the code generation schemes described below.

In doing code generation from a high level modeling language such as ASCEND there are a number of problems that need to be solved:

- how to generate the code efficiently, in particular the code necessary for (at least) first derivative calculations.
- how to add the compiled generated code to the system.
- how to access the generated code while maintaining all the functionality of the modeling language. In other words, an ASCEND model that uses externally compiled code should behave no differently than one using interpreted code with respect to name space and the operations of IS_REFINED_TO and ARE_THE_SAME, as well as the execution of the initialization procedures.
- how to automate the process of code generation.

To solve these problems a code generation module was added to the system which generates the code for function evaluations and generates the symbolic derivatives for all incident variables on a relation. One function is generated for function evaluation and one for gradient evaluation for each relation in the problem submitted to the code generation routines. The gradient routines return a vector of derivatives for each variable that matches the ASCEND compiler's definition of a variable, whereas the function evaluation routines return a scalar which represents the residual of the relation.

An efficient memory management routine was developed for generation of the symbolic derivatives to eliminate the need to call upon the underlying operating system to deal with the sub expression swell when doing symbolic derivatives. This code also performs sub expression elimination and constant folding wherever possible. The optimizations performed are conservative due to the loose definition of a variable by the ASCEND III compiler. In ASCEND IV, these optimizations may be much more aggressive due to the introduction of a dedicated SOLVER_REAL instance type and a **constant** type.

The code is generated in a format that is fully compatible with the requirements of the external glass-box relations discussed in Section 4.3.4.

In order to be able to use the generated code in a manner that is transparent to the system, a new language construct called the PATCH[3] was developed. Essentially a PATCH is a fragment of valid ASCEND IV code which carries the following information

- the *original* type specification of the model that was submitted for code generation.
- a collection of **glassbox_relation** statements where the **optional_scope** specifier is given.

The ASCEND IV code fragment in Example 23a shows the original model definition and a portion of the automatically generated PATCH file in Example 23b. Considering the example, to create an instance of MODEL **test_flash** *but using the compiled code representation of relations*, the PATCH file

3. The purpose of the PATCH is almost identical to the Unix utility of the same name.

containing the definition for **fast_flash** should be read in and the instantiation routines should be invoked with the patch option.

Internally, the following algorithm is used when compiling with the patch option:

- given the name of a PATCH, look up its *original* type definition. If it does not exist then abort.
- perform a partial instantitation of the original type definition, creating all instance structures but only creating stubs for the relations.
- process each **glassbox_relation** statement in the patch definition, using the standard glass-box instantiation algorithm but using the **optional_scope** reference to locate and place the relations.
- process any other additional statements in the PATCH definition.
- finish up normal instantiation.

```
MODEL VLE_flash(components : set OF symbol);
  (* other statements ommitted for clarity *)

    FOR i IN components CREATE
       f_def[i]:
        flash.f[i] = flash.Ftot * flash.y[i];
    END;

END VLE_flash;

MODEL test_flash;

  components IS_A set OF symbol;
  components := [‘a’,’b’,’c’];
  flash IS_A VLE_flash(components);

END test_flash;
```

(a)

```
IMPORT fastflash_Init FROM libfastflash;

PATCH fast_flash FOR test_flash;

    f_def[‘a’]: fast_flash(f[‘a’], Ftot, y[‘a’]; 1) IN flash;
    f_def[‘b’]: fast_flash(f[‘b’], Ftot, y[‘b’]; 2) IN flash;
    f_def[‘c’]: fast_flash(f[‘c’], Ftot, y[‘c’]; 3) IN flash;

END fast_flash;
```

(b)

EXAMPLE 23 The PATCH

The other allowable statements in a PATCH definition include default assignment statements which allows capture of the state of the instance at the time of code generation. The processing time of a PATCH is lower compared to the semantic analysis of the original type definition, because the processing time for a glass-box relation is much lower than for token relations.

## 4.6 OTHER

In addition to the code generation format necessary to support the PATCH, a number of different formats have been found to be of utility and are thus incorporated into system. These include

- planar ASCEND: an ASCEND model which represents a flattening of the ASCEND instance hierarchy.
- black-box C code: function and gradient evaluation routines, sparse matrix incidence pattern, lower and upper bounds, scaling and an initial point. This format accepts an specification of input and output variables as well as parameters. It is thought that the format of this generated code is an efficient representation for sharing nonlinear test problems.
- a GAMS [BKM88] input file, used mainly for debugging the interface to optimizes attached to the ASCEND IV system.
- a Mathematica [Wol88] input file, used extensively in checking the correctness of the symbolic derivative code developed for many of the above code generation schemes.

# 4.7 APPENDIX

## 4.7.1 RELATION TERM DATA STRUCTURES

```
struct RelationReal{
  double value;
  CONST dim_type *dimensions;
};

struct RelationVar {
  unsigned long varnum;        /* index into the variable list */
};

struct RelationFunc {
  CONST struct Func *fptr;
  struct relation_term *left;
};

union TermUnion {
  long ivalue;                  /* integer value */
  struct RelationReal r;        /* real value */
  struct RelationVar var;       /* vars */
  struct RelationFunc func;     /* funcs */
  struct UnaryOp {    /* unary minus, plus -- used for infix only */
    struct relation_term *left;
  } uni;
  struct BinaryOp {     /* binary, plus,minus,divide,times,power */
    struct relation_term *left;
    struct relation_term *right;
  } bin;
};

struct relation_term {
  enum Expr_enum t;             /* type of term */
  union TermUnion u;
};
```

### 4.7.2 NEW RELATION DATA STRUCTURE

```
struct TokenRelation {
  struct gl_list_t *lhs, *rhs;                  /* postfix */
  struct relation_term *lhs_term, *rhs_term;    /* infix */
  REFCOUNT_T ref_count;
};

struct OpCodeRelation {
  int *lhs, *rhs;                               /* array of opcodes */
  int *args;
  int nargs;
  double *constants;                            /* array of reals */
  REFCOUNT_T ref_count;
};

struct GlassBoxRelation {
  struct ExternalFunc *efunc;
  int *args;                   /* an array of indexes into the varlist */
  int nargs;
  int index;                   /* the *external* index of this relation */
  REFCOUNT_T ref_count;
};

struct BlackBoxRelation {
  struct ExtCallNode *ext;     /* external call info */
  int *args;                   /* an array of indexes into the varlist */
  int nargs;
  REFCOUNT_T ref_count;
};

union RelationUnion {
  struct TokenRelation token;
  struct OpCodeRelation opcode;
  struct GlassBoxRelation gbox;
  struct BlackBoxRelation bbox;
};

struct relation {
  union RelationUnion *u;      /* intentionally a pointer */
  double residual;
  double multiplier;
  double nominal;
  struct gl_list_t *vars;
  dim_type *d;
  enum Expr_enum relop;        /* type of constraint */
};
```

### 4.7.3 EMBEDDED BLACK-BOX RELATION

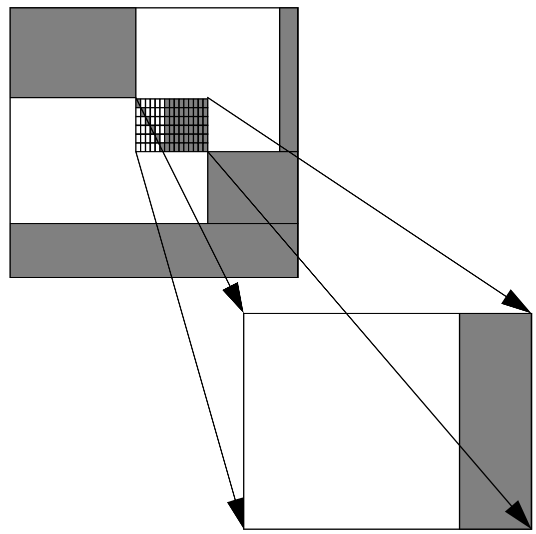


FIGURE 18 Exploded View of Black-Box matrix

## 4.8 REFERENCES


[BKM88] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user's guide*. Scientific Press, 1988.

[WHMW79] A. W. Westerberg, H. P. Hutchinson, R. L. Motard, and P. Winter. *Process Flowsheeting*. Cambridge University Press, 1979.

[Wol88] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley Publishing, 1988.

# CHAPTER 5 LINEAR ALGEBRA

## 5.1 ABSTRACT

Some of the important aspects of solving large sparse linear systems of equations of the form **Ax = b** are reviewed, in particular sparse matrix analysis. The decomposition schemes often used to tackle these systems are also examined and an attempt is made to explain the reasons for their inconsistent efficiency. A new decomposition algorithm for sparse matrix analysis and factorization is presented which seeks to overcome the difficulties. It is tested on problems with order up to 80,000 and over 390,000 nonzeros. The algorithms proposed can make many existing **LU** factorization algorithms much faster, allowing them to be competitive with the best algorithms available.

## 5.2 INTRODUCTION

At the base of most numerical techniques is the solution of a linear system of equations of the form **Ax = b**. For many systems the matrix **A** is a sparse real *nxn* matrix and may or may not possess desirable properties such as structural and numerical symmetry, positive definiteness, or diagonal dominance. Often this linear system of equations has to be solved with a different matrix **A** (structurally and numerically), and for different right hand sides **b**. The cost of solving these linear systems in many cases dominates the cost of solving the problem in which they are embedded. This cost tends to grow at least quadratically with $n$.

Using the model of Duff et al, [DER89], solving a system of linear equations may be broken down into *analyze, factor,* and *solve* phases. The discussion here will be limited to the *direct methods* for solving linear systems rather than the *iterative* techniques, as this work seeks to address general problems which may not have any of the desirable properties that make the latter class of solution techniques applicable. It is well known that at the end of a solution of a system of linear equations by direct methods that multiple right hand sides (possibly transposed) may then be solved with minimal additional work. This is not true of the iterative techniques. In addition, only systems where the matrix **A** is an *nxn* real unsymmetric matrix will be considered.

In the analyze phase, the matrix **A** is normally first subjected to a maximal transversal algorithm [DER89] which seeks to find a zero free diagonal. If a maximal matching (otherwise known as an output assignment) cannot be found, then the matrix **A** is structurally singular. A sparsity preserving reordering (SPR) is normally applied to the matrix **A**. Some schemes employ the SPR *dynamically* while factoring the matrix, while others employ an *a priori* SPR and some form of pivoting to maintain numerical stability during factorization. Among the popular reordering schemes are nested dissection [Geo73], minimum degree [GL80], $P^4$ [HR72], SPK1 [SW84a], and the HP series of reorderings [LM77]. The Markowitz criteria, originally developed in 1957, and its variants are popular dynamic reordering schemes. Duff et al, [DER89] gives a detailed explanation of the Markowitz scheme.

Most of these SPRs have a complexity of at least $O(n\tau)$, where $n$ is the order of the matrix and $\tau$ is the number of nonzeros. The complexity of maximal matching algorithms is also $O(n\tau)$ although an algorithm has been presented by Hopcroft [HK73] which has somewhat better complexity. The multiplier for the SPRs is significantly greater than that for the maximal matching algorithms. Some analyze implementations subject the matrix to a *block lower triangular permutation* (BLT permutation), which will find any irreducible blocks which may be present in the matrix. The complexity of the block lower triangular permutation is $O(n) + O(\tau)$ , once a zero free diagonal has been obtained.

The factor phase then proceeds using some variant of Gaussian elimination. The popular variants include **LU** factorization, row-wise Gaussian elimination and the Rank Implicit schemes of Stadtherr et al, [SW84b]. The frontal and multi-frontal methods which were originally developed for diagonal or banded systems are being actively explored by a number of workers, [MS95] [CS95] [DD93], and the references contained therein. Some form of pivoting is used by most of the schemes to maintain numerical stability while factoring; this is a straight forward exercise in some methods and very difficult in others.

Once the lower and upper triangular factors have been found, the solution vector **x** may be obtained by a forward substitution and backward elimination against a right hand side vector **b**. For example, if **LU** factorization is used to factor the matrix **A**, then Equation 25 through Equation 27may used to solve for the solution vector **x**.

$$A = LU \tag{24}$$

$$LUx = b \tag{25}$$

$$Ly = b \tag{26}$$

$$Ux = y \tag{27}$$

The cost of solving a single right hand side **b** is proportional to the number of nonzeros in the **LU** factors. In practice this cost is significantly less than the analyze and factor phases. If, however, multiple right hand sides need to be solved, the cost may be non-trivial. Alvarado et al [ATE91] examine techniques for fast forward and backward substitution.

For most systems, the very expensive phases are the analyze and factor phases. The need to perform pivoting for numerical stability when dealing with general unsymmetric sparse matrices makes it difficult to put a tight *a priori* bound on the cost of the factorization. The **ma28** manual [Duf77] empirically found that it required $O\left(\tau^2/n\right)$ operations, where $\tau$ is the number of nonzeros in the *factored* matrix to do a factorization.

Decomposition techniques have been proposed in the literature to attempt to break the at least $O(n\tau)$ barrier associated with analysis and factorization.The results have been mixed, with most proponents of decomposition resorting to the apology:

> *Decomposition techniques, though often times slower than dealing with the problem as a whole, are necessary when the problem size becomes large enoug*h.

In addition, some of the decomposition schemes are not necessarily numerically stable [DER89].

Given this preamble, the rest of this chapter is organized as follows:

- The key components of analyzing and factoring a sparse matrix are discussed. This includes desirable matrix forms, sparsity preserving reorderings and pivoting for numerical stability.
- The decomposition schemes used in the literature are examined with the purpose of understanding why they give inconsistent behavior.
- A new analyze and factor algorithm is presented, which also uses decomposition but attempts to avoid the problems associated with other decomposition schemes. This algorithm is then tested on a number of large matrices and using a number of sparse linear algebra packages.

It will be shown that, where the new algorithm can be exploited, much lower reordering times and better reorderings are obtained with a consequent reduction in fill, operation count and factorization times. Finally other research issues in large scale linear algebra are discussed.

## 5.3 SPARSE MATRIX ANALYSIS

### 5.3.1 BLOCK LOWER TRIANGULAR FORMS

In solving a set of linear equations of the form **Ax = b**, it is well known that significant savings in computations can be made if the matrix **A** can be permuted to *block lower triangular form* (BLT).

**(28)**

$$PAQ = \begin{bmatrix} B_{11} & & & \\ B_{21} & B_{22} & & \\ . & . & . & \\ B_{N1} & B_{N2} & & B_{NN} \end{bmatrix}$$

A matrix with order $n$, which can be represented in this form with $n > 1$, is said to be *reducible*. Each diagonal submatrix $B_{ii}$ will be *irreducible*. The advantage of using block lower triangular forms is that only the diagonal submatrices need to be factored. Indeed, if a system of nonlinear equations $f(x) = 0, x \in \Re^n$, is to be solved and a block lower triangular form may be found from the sparse Jacobian matrix of the nonlinear system, then a Newton scheme need only be applied to irreducible blocks with $n > 1$. Functions with one unknown variable may be directly solved by some numerical root finding technique or symbolically.

Very fast algorithms exist for obtaining a block lower triangular form, provided that a maximum transversal exists. These include in the algorithms of Sargent and Westerberg [SW64] and Tarjan [Tar72]. The complexity has been shown to be $O(n) + O(\tau)$.

If the reducibility of the system is defined as $\gamma$ where

$$\gamma = \frac{B_{max}}{n} \tag{29}$$

$$B_{max} = \max_{i}(order(B_{ii})) \tag{30}$$

a measure of the coupling of a linear system may be obtained. Arguably, it is this metric which determines the work involved in solving a linear system **Ax = b**, rather than $n$ or $\tau$. $\gamma$ is bounded between 0 and 1, with higher values indicating greater coupling. The following data is extracted from a number of sources and gives an idea of the reducibility of different matrices.

**Table 5: Matrix Reducibility**

| Case | $n$ | $B_{max}$ | $\gamma$ | $\tau$ |
|---|---|---|---|---|
| A.2 | 541 | 541 | 1.0 | 4285 |
| A.10 | 1176 | 1174 | 1.0 | 18552 |
| A.16 | 2021 | 1500 | 0.742 | 7353 |
| A.19 | 4929 | 4578 | 0.93 | 33185 |
| B.1 | 37837 | 25351 | 0.67 | 412302 |
| B.2 | 77214 | 46328 | 0.60 | 633720 |
| B.3 | 176642 | 93620 | 0.53 | 986792 |
| B.4 | 396213 | 63394 | 0.16 | 3276552 |
| C.1 (isom a) | 33083 | 19995 | 0.61 | 105102 |
| C.2 (isom b) | 33083 | 21 | $6.35 \times 10^{-4}$ | 105102 |
| C.3 (4cols a) | 14388 | 2977 | 0.207 | 46321 |
| C.4 (4cols b) | 14388 | 12104 | 0.841 | 46396 |

The matrices and the reducibility data for group A were obtained from [DR93]. The B matrices and data were obtained from [TV96]. The C group of matrices were obtained from models developed within the ASCEND system.

The results for matrices isom.a and isom.b are especially remarkable as for a *single* change in the specification of the problem (the degrees of freedom were swapped around thus converting what was an initial value problem into a boundary value problem) the size of the largest block increased from 21 to 19995. In the context of chemical engineering problems, it is thought that $\gamma$ will tend towards 1, as plants become more coupled through heat integration.

### 5.3.2 SPARSITY PRESERVING REORDERINGS

Sparsity preserving reorderings (SPR) are used, either for minimizing fill in, or to confine fill in, when factoring a sparse matrix. The SPR algorithms are applied to the matrix either *a priori* or dynamically during factorization. The Markowitz criterion is a dynamic reordering scheme. Some of the *a priori* techniques that have been developed are:

- One-way dissection and nested dissection with has roots in finite-element sub-structuring, [Geo73]. This is related to tearing which will be discussed in some detail later.
- minimum degree and its multilevel variants.
- $P^4$ and $P^5$.
- SPK1 [SW84a], [Woo82].
- The HP series of reorderings.

A detailed discussion of sparsity preserving reordering algorithms will not be attempted here, and the above references as well as Chapter 7 of the excellent text by Duff et al, [DER89] should be consulted. It suffices to mention that sparse matrix reorderings fall into the class of NP-complete problems, thus the reordering algorithms are heuristic. The complexity of reordering varies with the different algorithms but has been found to be typically $O(n\tau)$ . For large matrices these reorderings becomes very expensive, while often failing to yield desirable results.

The SPK1 algorithm as described by Wood in his thesis is presented in the Appendix for completeness. It is representative of the other SPRs which have been successful when applied to small unsymmetric matrices. The important point to note about the algorithm is the use of column and row counts in decision making, particularly in the tie breaking criteria used in step 4. For large matrices with a lot of natural structure and repeated structures, there will be many rows and columns with *identical* row and column counts; any incidence count tie breaking rules then quickly fail.

In addition, most SPR algorithms have some *greedy* element (as a global optimization is prohibitively expensive). Greedy heuristics along with failure of tie breaking rules very quickly lead to bad decisions which propagate and lead to poor reorderings. Empirically it has been observed that the best the upperbound on problem size for SPRs such as SPK1 may be as low as 900 to 1200 rows.

Stadtherr and Wood [SW84a] give detailed comparisons of several reordering techniques applied to unsymmetric matrices. They report that their BLOKS algorithm (one motivated by chemical engineering problem structures) which restricts reordering to particular regions of the matrix, consistently gave the lowest reordering times. The reordering times for SPK1 for a few problems are shown in Table 6. These times are considered to be prohibitive.

**Table 6: Reordering Times**

| | n | nnz | Time (sec) |
|---|---|---|---|
| 4Cols | 12456 | 44800 | 20.89 |
| 10Cols | 31140 | 115830 | 119.98 |
| Isom_30K | 19995 | 105102 | 27.73 |

The Markowitz criteria has also been observed to give very long run times on large matrices [DR93]. **ma28** uses a Markowitz search whereas the second generation code **ma48** uses a Zlatev type search [DR93].

### 5.3.3 PIVOTING FOR NUMERICAL STABILITY

During a **LU** factorization, it is almost always necessary to perform some form of *pivoting* to maintain numerical stability. If a zero or near zero diagonal element is encountered at any stage of the **LU** factorization algorithm, it is necessary to exchange that element with an element of larger value. In the limit of *full pivoting*, at each step, the largest element in the remaining submatrix would be chosen and permuted to become the new pivot. In *partial pivoting,* the largest element in each row (or column) is chosen as the new pivot. Pivoting, full or partial, can be an expensive process. In the case of full pivoting the entire remaining submatrix would need to be searched for the largest element. This process would have to be repeated for every step of the factorization. In addition, the desirable structures created from an *a priori* sparsity preserving reordering may be destroyed by pivoting. To offset these difficulties *threshold pivoting* is normally employed. Threshold pivoting is a relaxation of partial pivoting that requires at every step the following relationship be satisfied:

$$\left|a_{kk}^{k}\right| \geq u\left|a_{kj}^{k}\right|, \forall j, j > k \qquad \textbf{(31)}$$

where $u$ is a suitable value in the range $0 < u \leq 1$. If $u = 1$ then the partial pivoting criteria as shown below is obtained:

$$\left|a_{kk}^{k}\right| \geq \left|a_{kj}^{k}\right|, \forall j, j > k \qquad \textbf{(32)}$$

Equation 31 essentially states that a pivot is sought which is at least some fraction, of the largest element to the right of the diagonal in the current row. If the current diagonal element does not satisfy this criterion, then a column swap must be performed to place an element which does satisfy this criteria in the diagonal position. A value of $u = 0.1$ is used by many workers.

### 5.3.4 TEARING

The solution of problems by tearing is an approach by which part of the given problem is *torn away,* so that the remaining subproblems can be *analyzed* independently [Haj80]. Most workers who have employed tearing have pushed this concept further; they solve the now independent subproblems and then attempt to combine their solutions with the torn-away part in order to obtain the solution for the overall problem. The literature on tearing (or decomposition) is essentially divided into work concerned with how to best tear a problem so that it possesses certain desirable properties and work concerned with solving a problem which has been torn. A detailed discussion of the former is presented in Chapter 6.

Tearing is also known by the name *diakoptics* in the electrical engineering literature and was apparently first introduced by Kron in 1951 [Kro63]. It is a widely used technique in chemical engineering practice where it has been applied to solving flowsheets in a *sequential modular* manner [WHMW79]. Sequential modular solving amounts to tearing at the nonlinear level of the problem. Hajj claims that tearing and partitioning are essentially identical operations and that tearing is just a special form of partitioning a problem. The benefits of tearing are potentially large. The reasons often cited are:

- the smaller subproblems can help to break the $O\left(n^2\right)$ and worse behavior associated with many systems by making $n$ small.
- the size of the problem, even with sparse matrix techniques, requires that the problem be decomposed to accommodate limited computing resources.
- having the problem decomposed allows for parallel processing of the sub-problems.
- latency can be exploited; this is peculiar to time-analysis of electrical circuits, where the slowly changing subproblems need not be re-analyzed at each time step.

Many approaches have been used in the literature to obtain different structures from tearing. These include performing tearing so as to obtain a *bordered block diagonal* (BBD) problem, where there are many small blocks of even size and a narrow border. The resulting matrix is then factored using *block factorization schemes* such as the $F_1$, $F_2$, or $F_3$ factorization schemes of George, [Geo74], [Haj80], [Ior90]. Other workers have performed tearing recursively, obtaining multiple nested borders. This results in a *recursive bordered block diagonal* (RBBD) matrix [Vla85,] [NNA90], [NNA91], [Mor89]. The problem is then solved using a recursive block factorization scheme.

All of these techniques depend upon the formation of the Schur complement and so they require that the border (or borders in the case of the RBBD schemes) be kept narrow and the diagonal blocks nonsingular. A notable exception is the *balance bordered decomposition* scheme of Zecevic and Siljak [ZS94], whose algorithm is designed for parallel processing only. They thus seek a single border which is of the same size as each of the diagonal blocks.

The success of the above techniques has been mixed. Vlach reports higher fill than without tearing. Asai et al. report *much lower* fill. Almost all workers find that there is an *increase* in factorization time when using single processors, and most

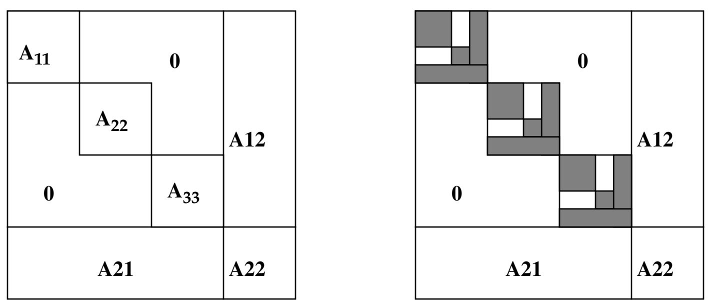



FIGURE 19 Bordered Block Diagonal and Recursive Bordered Block Diagonal Matrices.

resort to the *apology of decomposition* given on page 104. Another open question is the issue of the numeric stability of block factorization schemes as aptly raised by Duff et al. [DER89], as well as the applicability of the algorithms to large unsymmetric matrices. The matrices tested in the literature reviewed in this section are of relatively low order and often structurally and numerically symmetric.

### 5.3.5 BLOCK FACTORIZATION

The mixed results of decomposition schemes are puzzling. In this section an attempt is made to explain why these schemes do not perform uniformly, in particular why block factorization schemes based on the formation of the Schur complement fail. George, describes the $F_1$, $F_2$ and $F_3$ factorization schemes as follows:

$F_1$ factorization

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \tag{33}$$

$F_2$ factorization

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & \\ A_{21} & \bar{A}_{22} \end{bmatrix} \begin{bmatrix} I & \bar{U}_{12} \\ & I \end{bmatrix} \tag{34}$$

$F_3$ factorization

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \tilde{L}_{21} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ & \tilde{A}_{22} \end{bmatrix} \tag{35}$$

Consider the $F_2$ factorization. The following systems of linear equations need to be solved:

$$A_{11}\bar{U}_{12} = A_{12} \tag{36}$$

$$\bar{A}_{22} + A_{21}\bar{U}_{12} = A_{22} \tag{37}$$

If the region **A11** is very reducible (see Figure 19), then each of the diagonal blocks may be factored separately. If multiple processors are available, then these may done in parallel. This makes block factorization an attractive approach. However, factorization of **A11** is not necessarily numerical stable. Implicit in the factoring of **A11** is the inability to perform pivoting in the border columns **A12**. It is then possible for the matrix **A11** to be numerically singular whereas the overall problem is nonsingular. This problem is accentuated with the RBBD schemes due to the presence of multiple borders, in which pivots may *not* be selected.

The matrix $\bar{U}_{12}$ needs to be formed (implicitly or otherwise) as well as the Schur complement $\bar{A}_{22}$. Now during factorization $\bar{A}_{22}$ will become denser than **A**. If the order of $\bar{A}_{22}$ is low, dense code may be used to factor it. If the order is high, then sparse code must be used. However, with up to an order of magnitude greater density, the reordering of $\bar{A}_{22}$ will be very expensive. If it is not reordered then it could fill in completely. The RBBD schemes control the size of any given border, thus offsetting the above dilemma, but require significantly more complicated data structures to manage nested borders.

## 5.4 NEW ALGORITHMS

### 5.4.1 MOTIVATION

In the previous sections the analyze, factor and solve models were explained. It is clear that the analyze and factor phases are the most expensive. A number of observations have been made about these processes, motivating the development of the algorithms which will be presented.

- Sparsity preserving reorderings (SPR) are expensive. For large matrices the analyze time can be significantly greater than the factorization time because of the $O(n\tau)$ or worse complexity.
- SPR are based on heuristics. In particular most of these algorithms employ row and column counts in their decision making process. They also depend on tie breaking rules. For large problems, or problems with a lot of structure, the tie breaking rules quickly fail. Bad decisions are made which then propagate throughout the rest of the reordering.

These factors lead to two (2) main results: slow reorderings and bad reorderings.

Bad reorderings lead to much greater fill, resulting in increased memory for factorization, much higher operation count, longer factorization times and, perhaps worst of all, poor solution residuals resulting from numerical round-off and bad pivot selection.

Additionally, the following conjecture is made: *The cost of factoring an irreducible matrix **A** should be only nominally more expensive than that of factoring a matrix bordered block diagonal matrix **A′**, where the region **A11** is highly reducible*.

### 5.4.2 STATEMENT OF ALGORITHM

Given the above observations and the difficulties associated with block factorization the following simple algorithm was developed which uses as input an irreducible matrix **A**.

1. Attempt to output assign the matrix **A** using a maximal transversal algorithm. If a maximal transversal cannot be found, **quit**; the matrix is structurally singular.
2. Find a set of tear variables and/or connection equations by some suitable technique, such that, when these columns/rows are removed from the matrix **A**, the resulting submatrix **A11** is highly reducible.
3. For each variable in the tear set, perform a symmetric permutation of the variable and its *matching* row to the lower right hand corner of the matrix. This will create a border corresponding to **A12**, **A21** and **A22**.
4. Using a suitable algorithm find independent diagonal blocks within **A11**. If any of the *induced blocks* are above a predetermined size $n_{max}$, denote this region as **A** and **goto** 2. The choice of algorithm will result in a *recursively bordered block lower triangular* (RBBLT) matrix or a *recursively bordered block diagonal* (RBBD) matrix.
5. Reorder each resulting induced block using a sparsity preserving reordering algorithm.
6. Perform a **LU** factorization on the *entire* matrix, using a *modified threshold pivoting strategy*.
7. **stop**.

There are number of features of the algorithm which require explanation. If a matrix **A** is irreducible and the order $n$ is large, then any attempt to reorder the entire matrix will be confronted with the difficulties described in the discussion of sparsity preserving reorderings (SPR). If, however, the SPR can be restricted to certain regions of the matrix which are below some threshold size $n_{max}$ known to give good reorderings, then these difficulties may be overcome.

The algorithm described attempts to find these regions to which reordering may be restricted, by *inducing separability* of the matrix **A**. This is done in Step 2 by removal of certain rows and their *matching* columns leaving the induced region **A11** reducible. The matchings are a product of the output assignment done in Step 1, which is necessary to establish the structural rank of the problem. Choosing the rows and columns that are to be removed is a tearing problem. Algorithms for choosing such rows and columns will be presented in detail in Chapter 6.

Step 3 involves finding independent diagonal subblocks. The purpose of this step is two-fold. Firstly, by requiring that the induced subblocks of **A11** (such as $A_{11}$, $A_{22}$ and $A_{33}$ in Figure 19) be disjoint, then any reordering applied to one block will not affect the reordering of another block. Secondly, it establishes whether any such block is above the threshold $n_{max}$; if so then the algorithm may be applied recursively to these blocks. This step allows some flexibility in the algorithm that may be used to find the independent blocks. If a block lower triangular permutation (BLT) algorithm as described in a previous section is used, a recursive bordered block lower triangular (RBBLT) or an approximately recursive bordered block bordered (RBBD) matrix will be obtained. If a partitioning algorithm as described by Sangiovanni-Vincentelli et al. [SVLK77] is used at this phase, a RBBD matrix will be obtained. The independent diagonal submatrices found by a BLT algorithm will be irreducible.

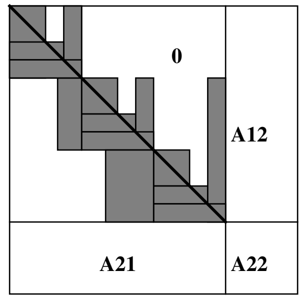



FIGURE 20 A recursively bordered block lower triangular matrix

The independence of the induced blocks may allow reorderings to be done in parallel, if multiple processors are available. In addition, different SPRs may be applied to different blocks.

Reordering times will be faster. If it is assumed that after the removal of a set of rows and their matching columns *m* blocks of equal size are induced, then a very simplistic analysis of the reordering time gives

$$m \cdot O\left(\frac{n}{m} \cdot \frac{\tau}{m}\right) = \frac{1}{m} \cdot O(n\tau) \tag{38}$$

which is an *m-fold* reduction over the undecomposed scheme.

These benefits are not without their cost; the border(s) **A21** and **A12** that are created by the removal of the rows/columns and their matching columns/rows respectively are not reordered. In principle the region(s) **A22** should be reordered if there are any superdiagonal elements. The above steps make up the analysis phase of the problem.

In the factorization phase the *entire* matrix is factored. The threshold pivoting criteria for maintaining numerically stability is modified to take into account the presence of multiple nested borders. At any stage of the factorization the column with the lowest index *l* (the column nearest to the diagonal) that satisfies the threshold pivoting criteria in Equation 31 is chosen as the new pivot. This ensures that structures developed during the analysis phase are disturbed as little as possible. Since pivoting is *always* allowed, the algorithm presented is no less numerically stable than regular sparse **LU** factorization. The block factorization schemes have no such guarantees. More elaborate pivoting schemes were considered, such as *on the fly column-splitting* which changes the matrix order, in order to preserve the structures found in the analysis phase. The difference in performance on the problems tested was negligible.

## 5.5 NUMERICAL EXPERIMENTS

### 5.5.1 TEST PROCEDURE

The algorithm presented in the previous section was tested on a number of matrices, primarily derived from chemical engineering problems. The classification and source of the matrices is given in the following table. For each matrix the source of the matrix, its order and number of entries, as well as a brief description is given. The very extensive test suite from the Harwell-Boeing collection [DGL92] could not be exploited as information required to assist the tearing selection is not available. In the following chapter a proposal is made for a new matrix format for communicating matrices and for an augmented application programming interface (API) necessary to allow efficient **LU** factorization.

**Table 7: Test Matrices**

| Case | Identifier | $n$ | $\tau$ | $\rho$ | $\gamma$ | Description |
|---|---|---|---|---|---|---|
| 1 | Isom_30K | 19995 | 105102 | 5.256 | 1.000 | Rigorous boundary value formulation of a pentane isomerization reaction with 200 time steps. 4 reacting species in 8 components. |
| 2 | 4Cols | 12456 | 44800 | 3.720 | 0.945 | 4 Mass balance distillation columns with 9 components. 30 trays per column, with the last column bottoms recycled to the first column. |
| 3 | 10Cols | 31140 | 115830 | 3.720 | 0.947 | As in 2, but with 10 columns. |

**Table 7: Test Matrices**

| Case | Identifier | $n$ | $\tau$ | $\rho$ | $\gamma$ | Description |
|---|---|---|---|---|---|---|
| 4 | BiqEquil | 8986 | 54389 | 6.046 | 0.440 | Rigorous distillation column with 30 trays, and 9 components. |
| 5 | PPP | 14698 | 64023 | 4.356 | 0.558 | Rigorous distillation column (propylene splitter) with 164 trays and 3 components. |
| 6 | Wood7 | 6858 | 33776 | 4.925 | 0.512 | Complex hydrocarbon flowsheet using rigorous thermodynamics, based on example 7 in [Woo82]. |
| 7 | Wood8 | 17509 | 132081 | 7.543 | 0.518 | Complex hydrocarbon flowsheet using rigorous thermodynamics and 15 components. Based on example 8 in [Woo82]. |
| 8 | Ethyl60 | 59080 | 294293 | 4.981 | 0.343[a] | Simplified model of ethylene plant. |
| 9 | Ethyl80 | 79554 | 398163 | 5.004 | 0.631 | As in 8 but with more stages and different degrees of freedom. |
| 10 | rdist3a | 2398 | 61896 | 25.812 | 1.000 | Reactive distillation column from J. Mallya [1994] |
| 11 | lhr04 | 4101 | 80755 | 19.19 | 0.861 | Light hydrocarbon recovery process from J. Mallya [1994] |

a. 3 large blocks, the largest of order 20316

All of the test matrices except rdist3a and lhr04 were obtained from models developed in the ASCEND system. Information to assist tearing for the rdist3a and lhr04 matrices was not available.

## 5.5.2 CODES

In order to test the algorithms proposed, five (5) direct (rather than iterative) sparse matrix solvers were used. These are **rankikw**, **LU1SOL**, **ma28**, **ma48** and **umfpack1.0**. A brief description of the codes tested is given in the next few sections.

### 5.5.2.1 rankikw

**rankikw** is an implementation of the RANKI factorization algorithm which is part of the ASCEND system and is based on the work by Stadtherr and Wood

[SW84b]. The underlying sparse data structures were coded by Karl Westerberg, Joseph Zaher and Benjamin Allan. The reordering algorithm is the **SPK1** algorithm of Stadtherr and Wood as implemented in the ASCEND system. Development continues on this code.

#### 5.5.2.2 LU1SOL

**LU1SOL** is a very efficient implementation of a rowwise Gaussian elimination coded by Stadtherr and Wood. This code does not save the **LU** factors but solves immediately for the solution vector **x**. It was developed in 1984. For the tests, the factorization code was culled from the spar2pas package and fed with a **SPK1** reordered matrix from the **rankikw** driver.

#### 5.5.2.3 ma28

**ma28** [Duf77] is a very popular code from the Harwell Subroutine Library and uses standard Gaussian elimination with pivoting controlled by the Markowitz criteria. Pivots must satisfy a threshold pivot requirement. The number of rows to be searched in establishing the Markowitz counts can be limited by a parameter NSRCH. This code has many options and only the defaults were used except for the NSRCH parameter. The default value for NRSCH is 32678. The factorization time for this code should be compared with the *total* reorder and factorization for the **rankikw** and **LU1SOL** codes because of the Markowitz based pivoting. This code has provision to do a *fast factorization*, in which the pivot sequences from a previous factorization is used.

#### 5.5.2.4 ma48

**ma48** [DR93] was designed to replace **ma28**. It uses columnwise Gaussian elimination with capability to switch to dense code at the later stages of factorization. It is fundamentally different from **ma28** in having distinct analyze and factor phases. Like **ma28**, this code has many options inclusive of a NRSCH parameter. The default for NRSCH in **ma48** is 3. A fast factorization mode is provided.

#### 5.5.2.5 umfpack1.0

**umfpack1.0** is the newest of the codes tested, being developed in 1995. It is uses an unsymmetric multi-frontal code algorithm developed and implemented by Davis and Duff [DD93]. It uses only dense kernels and uses a dynamic Markowitz search for pivot selection, and the search may be limited with an NSRCH parameter as in the case of **ma28**. The default is 4. It does not use an *a priori* reordering. Like **ma28**, it has a fast factorization mode. The times for this code are directly comparable with the **ma28** code.

The **rankikw** code is implemented in the C language and can make full use of the dynamic memory allocation features of the language. In addition this code uses *element pooling* so as not to have to make expensive calls on the operating system for memory. The other codes are implemented in FORTRAN. **ma28**, **ma48** and **LU1SOL** had their memory allocated from the C based **rankikw** driver. **umfpack1.0** was called with its own FORTRAN driver. All of the codes were provided with as much memory as recommended by their documentation and then some, in most cases an order of magnitude higher.

For **LU1SOL** and **rankikw**, two (2) sets of numbers are reported. The first is the data for these codes using the standard reordering and factorization schemes. The second set are the best times for these codes obtained using the new tearing and reordering (TEAR_DROP) algorithm. A value of 1000 or 2000 for $n_{max}$ was used for the TEAR_DROP algorithm. For **ma28** and **ma48** three (3) sets of raw data are reported. These correspond to values for NSRCH of 4, 10 and 1000 respectively. During preliminary runs with **ma28** using the default NSRCH parameter of 32678, the run times were horrendously long. It was then decided to investigate the effect of the NSRCH parameter on the **ma28** and **ma48** codes. However, for the data summaries, only the data for a NSRCH of 4 is given.

The timing results are cpu times in seconds obtained on a HP9000/715 with a maximum available memory of 256 MB. Wherever available the following information is reported:

- reordering time (cpu seconds).
- factoring time (cpu seconds).
- the number of elements in the factored matrix.
- operation count.

### 5.5.3 TEST RESULTS

In comparing the results, the eventual application of the codes is important. In the context of solving an *nxn* system of nonlinear equations using Newtons method or some variant, it is well known that approximately 7 to 10 Newton iterations are normally required for solution when starting from a good initial point. From an excellent initial point (as in slightly perturbed systems, which occurs during numerical integration), convergence can be achieved in as few as 2 - 3 Newton iterations. For the former case the cost of a SPR can be amortized over the 7 - 10 steps. Due to the significant changes in data over a small number of iterations, it is unlikely that a pivot sequence can be used between factorizations. For this problem class, codes that use *only* dynamic pivoting such as **ma28** and **umfpack1.0** are less attractive. In the case of sequential numerical integration with many iterations, (it is not uncommon to have hundreds of iterations, with slowly changing data), the cost of reordering may be negligible compared to the total factorization time. The slowly changing nature of data also suggests that a pivot sequence may safely be reused (with appropriate safeguards), which favors a code that has these capabilities. The detailed results are given in Section 5.7.2.

**Table 8: Normal Reorder vs. TEAR_DROP**

| | **LU1SOL** & **SPK1** | | | **LU1SOL** & TEAR_DROP | | |
|---|---|---|---|---|---|---|
| | reorder | factor | total | reorder | factor | total |
| Isom_30K | 27.73 | 16.35 | 44.08 | 5.29 | 0.87 | **6.16** |
| 4Cols | 20.89 | 22.39 | 43.28 | 2.53 | 2.93 | **5.46** |
| 10Cols | 119.98 | 114.78 | 234.76 | 9.81 | 7.27 | **17.08** |
| BigEquil | 1.9 | 4.7 | 6.60 | 0.46 | 0.49 | **0.95** |
| PPP | 10.27 | 0.41 | 10.68 | 1.9 | 0.38 | **2.28** |
| Wood7 | 1.5 | 0.24 | 1.74 | 0.42 | 0.18 | **0.60** |
| Wood8 | 13.38 | 26.47 | 39.85 | 2.34 | 4.36 | **6.70** |
| Ethyl60 | 50.67 | 13.22 | 63.89 | 8.78 | 2.31 | **11.09** |
| Ethyl80 | 204.94 | 32.35 | 237.29 | 15.49 | 3.95 | **19.44** |

In Table 8 the performance of the **LU1SOL** code with the standard **SPK1** reordering and the TEAR_DROP algorithm is shown. The TEAR_DROP drop times include all the time taken for tearing and reordering. In all cases the analyze and factor times are faster with the TEAR_DROP algorithm, in some instances by an order of magnitude. The number of fills and the operation counts are also always lower.

Similar results are obtained for the **rankikw** code (see Table 9). However, the improvement in factorization times was not as significant nor as consistent as the **LU1SOL** code. This is largely due to an expensive relabelling operation intrinsic to the RANKI algorithm.

**Table 9: Normal Reorder vs. TEAR_DROP**

| | **rankikw** & **SPK1** | | | **rankikw** & TEAR_DROP | | |
|---|---|---|---|---|---|---|
| | analyze | factor | total | analyze | factor | total |
| Isom_30K | 27.73 | 37.55 | 65.28 | 5.45 | 40.61 | **46.06** |
| 4Cols | 20.89 | 14.51 | 35.40 | 2.53 | 9.57 | **12.10** |
| 10Cols | 119.98 | 96.03 | 216.01 | 9.81 | 51.8 | **61.61** |
| BigEquil | 1.9 | 6.60 | 8.50 | 0.49 | 2.76 | **3.25** |
| PPP | 10.27 | 10.87 | 21.14 | 1.89 | 7.73 | **9.62** |
| Wood7 | 1.51 | 1.16 | 2.67 | 0.31 | 1.10 | **1.41** |
| Wood8 | 13.07 | 38.8 | 51.87 | 2.34 | 13.4 | **15.74** |
| Ethyl60 | 50.67 | 55.15 | 105.82 | 7.14 | 53.37 | **60.51** |
| Ethyl80 | 206.52 | 386.19 | 529.71 | 15.65 | 270.68 | **286.33** |

In Table 10, a comparison is made among the analyze and factor times for the five codes tested. In this table the *total* analyze and factor times are reported. For the codes which have a distinct analyze phase (**rankikw**, **LU1SOL** and **ma48**) the analyze time is provided in parentheses. A direct comparison of the codes is not intended but perhaps inevitable. **ma48** is the fastest of the codes for both a one-off factorization and as a linear algebra code to be used as a subproblem in solving a nonlinear system of equations. The **LU1SOL** code using the TEAR_DROP analyze is the second fastest code using the same criteria. Ignoring the **rankikw** code, all codes are within a factor of 4 of the fastest code.

One conclusion that may be drawn is that the TEAR_DROP algorithm, when applied to a code that can take advantage of better *a priori* reorderings (such as **rankikw** and **LU1SOL**) can make that code competitive with state of the art codes.

**Table 10: Analyze and Factor Time summary[a]**

| | **rankikw** & TEAR_DROP | **LU1SOL** & TEAR_DROP | **ma28** NSRCH = 4 | **ma48** NSRCH = 4 | **umfpack1.0** |
|---|---|---|---|---|---|
| Isom_30K | (5.45) 46.06 | (5.45) 6.16 | 4.56 | **(1.32) 1.67** | 2.64 |
| 4Cols | (2.91) 12.48 | **(2.53) 5.46** | 6.47 | (6.28) 8.24 | 5.97 |
| 10Cols | (9.81) 61.61 | **(9.81) 17.08** | 18.85[b] | (22.65) 29.49 | 18.35 |
| BigEquil | (0.49) 3.25 | (0.46) 0.95 | 0.88 | **(0.39) 0.54** | 1.59 |
| PPP | (1.89) 9.62 | (1.90) 2.28 | 1.42 | **(0.45) 0.59** | 2.08 |
| Wood7 | (0.31) 1.41 | (0.42) 0.60 | **0.29** | (0.25) 0.34 | 1.10 |
| Wood8 | (2.34) 15.74 | (2.34) 6.70 | **3.44** | (2.79) 3.62 | 5.03 |
| Ethyl60 | (7.14) 60.51 | (8.78) 11.09 | 5.69 | **(2.52) 3.22** | 7.47 |
| Ethyl80 | (15.65) 286.3 | (15.49) 19.44 | N/A | **(4.24) 5.25** | 9.87 |

a. the numbers in parentheses are the *a priori* analyze times where applicable.
b. **ma28** with NSRCH = 10, actually had a lower factorization time.

Another comparison that is worthwhile is the performance of a code that uses traditional sparse matrix technology (**LU1SOL**) versus a code that uses multi-frontal techniques. The argument for multi-frontal schemes is their ability to make use of dense kernels and to avoid the indirect addressing associated with traditional sparse codes [DD93], [ZMDS]. It is claimed by the proponents of multi-frontal schemes that it is this indirect addressing that results in the poor performance of traditional sparse codes. The comparable performance of **LU1SOL** with good reorderings against **umfpack1.0** suggests that the ill effects of indirect addressing may be severely overestimated (at least on workstations with similar architecture to the HP9000/715)

Hajj, [Haj80] defines tearing as *an approach by which part of the given problem is torn away so that the remaining subproblems can be analyzed independently.* It is then obvious that, by restricting analysis to a few rows via the NSRCH parameter, the Markowitz based codes are practicing *implicit tearing*. A simple interpretation of a restricted Markowitz scheme is that the matrix is divided into $m$ independent parts, where $m = n/\text{NSRCH}$. The saving in analysis time is then that given by Equation 38. The simple heuristic of restricting searches to a few row or columns has been very successful on the problems tested. However, very inconsistent results can sometimes be obtained, as shown for the lhr04 and rdist3a matrices (see Table 11)

**Table 11: Effect of NSRCH on ma48**

| | lhr04 | | | rdist3a | | |
|---|---|---|---|---|---|---|
| NRSCH | analyze | factor | total | analyze | factor | total |
| 4 | 35.38 | 20.99 | 56.37 | 53.07 | 33.66 | 86.73 |
| 10 | 33.49 | 18.49 | 51.98 | 49.54 | 26.63 | 75.84 |
| 40 | 29.14 | 15.73 | 44.87 | 15.86 | 38.53 | 54.39 |
| 70 | 24.43 | 15.5 | **39.93** | 21.68 | 31.14 | **52.82** |
| 100 | **15.72** | 31.48 | 47.2 | **14.25** | 33.77 | 48.02 |
| 1000 | 62.27 | **7.12** | 69.39 | 62.24 | **20.38** | 82.58 |

A surprising result is the much higher fill that is incurred with the *a priori* reordering schemes using the **SPK1** reordering algorithm as compared to the dynamic[1] reordering schemes (see Table 12). This same trend is observed when comparing operation counts. This suggests that the **SPK1** algorithm may not be as efficient as originally thought, even when restricted to reordering blocks of size $n_{max}$. A definite extension of this work is re-examination of the above results with different reordering algorithms. This could include the minimum degree algorithm and even a *static* version of the Markowitz criteria; the reordering algorithm would assume that no numerical pivoting would take place and, at any

1. **ma28**, **ma48** and **umfpack1.0** use dynamic reordering.

given stage of the reordering, should be made to operate on a region with some maximum size $n_{max}$ which would need to be determined.

**Table 12: Nonzeros in Factors**

| | **rankikw** & TEAR_DROP | **LU1SOL** & TEAR_DROP | **ma28** | **ma48** | **umfpack1.0** |
|---|---|---|---|---|---|
| Isom_30K | 1.95e5 | 3.08e5 | **1.08e5** | 2.36e5 | 1.23e5 |
| 4Cols | 4.04e5 | 6.00e5 | **9.67e4** | 3.23e5 | 3.68e5 |
| 10Cols | 1.07e6 | 1.43e6 | **2.48e5** | 8.67e5 | 1.05e6 |
| BigEquil | 1.07e5 | 1.45e5 | **3.30e4** | 7.23e4 | 8.81e4 |
| PPP | 1.40e5 | 1.24e5 | **4.50e4** | 8.85e4 | 9.10e4 |
| Wood7 | 6.53e4 | 7.82e4 | 6.76e4 | **4.85e4** | 5.78e4 |
| Wood8 | 4.00e5 | 5.04e5 | **9.89e4** | 2.70e5 | 3.42e5 |
| Ethyl60 | 1.06e6 | 9.08e5 | **2.61e5** | 4.35e5 | 4.11e5 |
| Ethyl80 | 1.76e6 | 1.04e6 | N/A | 6.14e5 | **5.32e5** |

In fact, a major potential area of research lies in total re-evaluation of the sparsity preserving reordering technology. A careful examination of the work by Stadtherr and Wood [SW84a] shows that some of the reordering schemes developed were not tested because of their prohibitively long execution times. If, however, the cost of reordering can be controlled by restricting it to certain regions of a matrix, as is done in this work, then more sophisticated reordering schemes may be examined. The potential then exists for research into an entire class of reordering algorithms where each algorithm aggressively seeks to minimize operation counts but which should *never* be invoked on submatrices above a threshold size.

## 5.6 DISCUSSION

The difficulties with the current approaches to solution of large unsymmetric sparse systems of linear equations have been highlighted. Arguments have been presented that attempt to explain why the block factorization schemes have not been uniformly successful for the solution of these problems. An analyze/factor algorithm which uses decomposition only at the analysis phase was presented. The algorithm is simple and has been driven by some observations about the irreducibility of sparse matrixes. By temporarily inducing separability in the problem through tearing, sparse matrix reorderings can be applied to much smaller regions. This results in much better overall reorderings, much lower reordering times, and a reduction in fill, operation counts and factorization times. Numerical stability is preserved as the technique does not attempt to use block factorization schemes. The algorithm is competitive with current state of the art codes.

Perhaps the most important conclusion of this work is *tearing (implicit or explicit) must be done when dealing with large scale problems*. This tearing can be very effectively and safely done at the analysis phase, rather than at the factorization phase of sparse linear algebra.

## 5.7 APPENDIX

### 5.7.1 SPK1 ALGORITHM

The algorithm described below is the **SPK1** sparsity preserving reordering algorithm as described by Wood in his thesis, [Woo82]. It involves two auxiliary routines, forward and backward triangularization, which were not presented in that work but which may be found in [SW84a]. They are included here for completeness.

To forward triangularize

1. Search for a row with a row count of one. Remove that row and the column in which the one nonzero occurs and put it in the first open position in the reordered matrix. If all row counts are greater than one then go to 3.
2. Adjust the row counts to account for the removal of the column in step 1. Return to 1.
3. End.

To backward triangularize

1. Search for a column with a column count of one. Remove that column and the row in which the one nonzero occurs and put it in the last open position in the reordered matrix. If all column counts are greater than one then go to 3.
2. Adjust the column counts to account for the removal of the row in step 1. Return to 1.
3. End.

To reorder

1. Forward triangularize, backward triangularize.
2. Partition matrix, put irreducible blocks on block stack.
3. Process first block on stack, if no more blocks go to 8.
4. Select a spike and put it in the spike stack.
   - The spike is selected from a row with minimum row count, ties are broken by summing the column counts of the columns that intersect with rows of minimum row count; the row with the largest sum is selected.

   - The column with the smallest column count in the row selected is assigned to that row as the next pivot; the remaining columns in the row are placed in decreasing order of their column counts on the spike stack.

5. Forward triangularize
   - if a row with no nonzeros is found, assign it to the latest spike column entry on the stack, pop the spike column from the stack; if the row is the last row in the block go to 7.

6. Go to 4.
7. Pop block from the block stack and go to 3.
8. End.

### 5.7.2 TEST RESULTS

The following tables provide the raw data obtained from the comparisons of the various sparse linear algebra packages.

**Table 13: Isom_30K**

| n = 19995 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 27.73 | 37.55 | 65.28 | 18.07e6 | 2.517e6 |
| **rankikw** & TEAR_DROP | 5.45 | 40.61 | 46.06 | 5.91e5 | 1.952e5 |
| **LU1SOL** | 27.73 | 16.35 | 44.08 | 48.56e6 | 2.60e6 |
| **LU1SOL** & TEAR_DROP | 5.29 | 0.87 | 6.16 | 2.20e6 | 3.083e5 |
| **ma28** 4 | | 4.56 | 4.56 | 1.250e5 | 1.084e5 |
| **ma28** 10 | | 4.57 | 4.57 | 1.226e5 | 1.084e5 |
| **ma28** 1000 | | 9.89 | 9.89 | 1.189e5 | 1.081e5 |
| **ma48** 4 | 1.32 | 0.35 | **1.67** | 2.329e5 | 2.359e5 |
| **ma48**10 | 1.53 | 0.35 | 1.88 | 2.329e5 | 2.359e5 |
| **ma48** 1000 | 40.4 | 0.36 | 40.76 | 2.344e5 | 2.363e5 |
| **umfpack1.0** | | 2.64 | 2.64 | 2.57e5 | 1.235e5 |

**Table 14: 4Cols**

| n = 12456 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 20.89 | 14.51 | 35.4 | 10.36e6 | 8.498e5 |
| **rankikw** & TEAR_DROP | 2.91 | 9.57 | 12.48 | 3.96e6 | 4.039e5 |
| **LU1SOL** | 20.89 | 22.39 | 43.28 | 66.14e6 | 1.703e5 |
| **LU1SOL** & TEAR_DROP | 2.53 | 2.93 | **5.46** | 9.363e6 | 6.000e4 |
| **ma28** 4 | | 6.47 | 6.47 | 4.420e6 | 9.670e4 |
| **ma28** 10 | | 9.82 | 9.82 | 8.162e6 | 1.066e5 |
| **ma28** 1000 | | 22.72 | 22.72 | 5.622e6 | 1.010e5 |
| **ma48** 4 | 6.28 | 1.96 | 8.24 | 17.58e6 | 3.234e5 |
| **ma48**10 | 5.85 | 1.75 | 7.60 | 18.233e6 | 3.234e5 |
| **ma48** 1000 | 49.9 | 1.20 | 51.1 | 19.239e6 | 2.731e5 |
| **umfpack1.0** | | 5.97 | 5.97 | 27.3e6 | 3.675e5 |

**Table 15: 10Cols**

| n = 31140 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 119.98 | 96.03 | 216.01 | 82.23e6 | 2.9328e6 |
| **rankikw** & TEAR_DROP | 9.81 | 51.8 | 61.61 | 11.41e6 | 1.0651e6 |
| **LU1SOL** | 119.98 | 114.78 | 234.76 | 313.9e6 | 5.787e6 |
| **LU1SOL** & TEAR_DROP | 9.81 | 7.27 | **17.08** | 22.70e6 | 1.430e6 |
| **ma28** 4 | | 18.85 | 18.85 | 13.432e6 | 2.489e5 |
| **ma28** 10 | | 14.14 | 14.14[a] | 9.198e6 | 2.378e5 |
| **ma28** 1000 | | 71.96 | 71.96 | 6.338e6 | 2.225e5 |
| **ma48** 4 | 22.65 | 6.84 | 29.49 | 53.848e6 | 8.667e5 |
| **ma48**10 | 25.17 | 6.62 | 31.79 | 42.909e6 | 8.778e5 |
| **ma48** 1000 | 186.12 | 6.74 | 192.86 | 81.426e6 | 8.760e5 |
| **umfpack1.0** | | 18.35 | 18.35 | 90.0e6 | 1.050e6 |

a. The fastest time was attributed to **LU1SOL** based on reporting the times for NSRCH = 4.

**Table 16: BiqEquil**

| n = 8986 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 1.9 | 6.6 | 8.5 | 8.960e6 | 3.485e5 |
| **rankikw** & TEAR_DROP | 0.49 | 2.76 | 3.25 | 1.474e6 | 1.068e5 |
| **LU1SOL** | 1.9 | 4.7 | 6.60 | 13.64e6 | 3.452e5 |
| **LU1SOL** & TEAR_DROP | 0.46 | 0.49 | 0.95 | 1.599e6 | 1.447e5 |
| **ma28** 4 | | 0.88 | 0.88 | 5.121e5 | 3.298e4 |
| **ma28** 10 | | 1.01 | 1.01 | 5.447e5 | 3.372e4 |
| **ma28** 1000 | | 2.70 | 2.70 | 2.161e5 | 3.050e4 |
| **ma48** 4 | 0.39 | 0.15 | **0.54** | 7.205e5 | 7.231e4 |
| **ma48**10 | 0.51 | 0.17 | 0.68 | 1.106e6 | 7.761e4 |
| **ma48** 1000 | 11.33 | 0.13 | 11.46 | 4.944e5 | 6.826e4 |
| **umfpack1.0** | | 1.59 | 1.59 | 7.71e5 | 8.8069e4 |

**Table 17: PPP**

| n = 14698 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 10.28 | 10.87 | 21.15 | 1.650e6 | 3.332e5 |
| **rankikw** & TEAR_DROP | 1.89 | 7.73 | 9.62 | 5.970e5 | 1.397e5 |
| **LU1SOL** | 10.27 | 0.41 | 10.68 | 8.000e5 | 1.104e5 |
| **LU1SOL** & TEAR_DROP | 1.9 | 0.38 | 2.28 | 4.820e5 | 1.243e5 |
| **ma28** 4 | | 1.42 | 1.42 | 9.291e4 | 4.502e4 |
| **ma28** 10 | | 1.48 | 1.48 | 9.242e4 | 4.509e4 |
| **ma28** 1000 | | 13.0 | 13.0 | 7.807e4 | 4.326e4 |
| **ma48** 4 | 0.45 | 0.14 | **0.60** | 1.330e5 | 8.845e4 |
| **ma48**10 | 0.56 | 0.15 | 0.71 | 1.298e5 | 8.838e4 |
| **ma48** 1000 | 20.5 | 0.14 | 20.64 | 1.252e5 | 8.879e4 |
| **umfpack1.0** | | 2.08 | 2.08 | 2.300e5 | 9.103e4 |

**Table 18: Wood7**

| n = 6858 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 1.51 | 1.16 | 2.67 | 5.520e5 | 7.275e4 |
| **rankikw** & TEAR_DROP | 0.31 | 1.10 | 1.41 | 4.410e5 | 6.527e4 |
| **LU1SOL** | 1.5 | 0.24 | 1.74 | 5.190e5 | 7.822e4 |
| **LU1SOL** & TEAR_DROP | 0.42 | 0.18 | 0.60 | 4.450e5 | 6.762e4 |
| **ma28** 4 | | 0.29 | 0.29 | 6.957e4 | 2.133e4 |
| **ma28** 10 | | 0.34 | 0.34 | 7.041e4 | 2.122e4 |
| **ma28** 1000 | | 1.30 | 1.30 | 6.072e4 | 2.096e4 |
| **ma48** 4 | 0.25 | 0.09 | **0.34** | 1.671e5 | 4.855e4 |
| **ma48**10 | 0.29 | 0.08 | 0.37 | 1.781e5 | 4.849e4 |
| **ma48** 1000 | 8.96 | 0.08 | 9.04 | 1.375e5 | 4.731e4 |
| **umfpack1.0** | | 1.10 | 1.10 | 3.680e6 | 5.784e4 |

**Table 19: Wood8**

| n = 17509 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 13.07 | 38.8 | 51.87 | 5.340e7 | 1.249e6 |
| **rankikw** & TEAR_DROP | 2.34 | 13.4 | 15.74 | 8.900e6 | 3.996e5 |
| **LU1SOL** | 13.38 | 26.47 | 39.85 | 7.360e7 | 9.488e5 |
| **LU1SOL** & TEAR_DROP | 2.34 | 4.36 | 6.70 | 1.300e6 | 5.041e5 |
| **ma28** 4 | | 3.17 | **3.17** | 1.976e6 | 9.892e4 |
| **ma28** 10 | | 3.44 | 3.44 | 1.981e6 | 9.998e4 |
| **ma28** 1000 | | 15.63 | 15.63 | 1.747e6 | 9.385e4 |
| **ma48** 4 | 2.79 | 0.83 | 3.62 | 6.736e6 | 2.699e5 |
| **ma48**10 | 2.59 | 0.82 | 3.41 | 7.240e6 | 2.643e5 |
| **ma48** 1000 | 43.38 | 1.06 | 44.44 | 12.417e6 | 2.813e5 |
| **umfpack1.0** | | 5.03 | 5.03 | 1.600e7 | 3.416e5 |

**Table 20: Ethyl60**

| n = 59080 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 50.67 | 55.15 | 105.82 | 20.89e6 | 2.210e6 |
| **rankikw** & TEAR_DROP | 7.14 | 53.37 | 60.51 | 8.88e6 | 1.061e6 |
| **LU1SOL** | 50.67 | 13.22 | 63.89 | 34.00e6 | 1.743e6 |
| **LU1SOL** & TEAR_DROP | 8.78 | 2.31 | 11.09 | 6.03e6 | 9.075e5 |
| **ma28** 4 | | 5.96 | 5.96 | 1.321e6 | 2.614e5 |
| **ma28** 10 | | 6.13 | 6.13 | 1.195e6 | 2.589e5 |
| **ma28** 1000 | | 59.99 | 59.99 | 9.481e5 | 2.420e5 |
| **ma48** 4 | 2.52 | 0.70 | **3.22** | 7.223e5 | 4.354e5 |
| **ma48**10 | 3.00 | 0.69 | 3.69 | 7.256e5 | 4.353e5 |
| **ma48** 1000 | 87.96 | 0.75 | 88.71 | 7.197e5 | 4.383e5 |
| **umfpack1.0** | | 7.47 | 7.47 | 1.50e06 | 4.113e5 |

**Table 21: Ethyl80**

| n = 79554 | Analyze | Factor | Total | # Ops | nnz(F) |
|---|---|---|---|---|---|
| **rankikw** | 206.52 | 386.19 | 598.18 | 87.98e6 | 7.647e6 |
| **rankikw** & TEAR_DROP | 15.65 | 270.68 | 286.33 | 11.70e6 | 1.756e6 |
| **LU1SOL** | 204.94 | 32.35 | 237.29 | 70.30e6 | 4.446e6 |
| **LU1SOL** & TEAR_DROP | 15.49 | 3.95 | 19.44 | 8.550e6 | 1.043e6 |
| **ma28** 4 | N/A[a] | | | | |
| **ma28** 10 | | | | | |
| **ma28** 1000 | | | | | |
| **ma48** 4 | 4.24 | 1.01 | **5.25** | 1.055e6 | 6.136e5 |
| **ma48**10 | 4.85 | 1.02 | 5.87 | 1.066e6 | 6.136e5 |
| **ma48** 1000 | 134.74 | 1.10 | 135.84 | 1.026e6 | 6.146e5 |
| **umfpack1.0** | | | 9.87 | 2.080e6 | 5.312e5 |

a. N/A - no solution possible, apparently due to internal limitations of the code.

## 5.8 REFERENCES


[ATE91] F. L. Alvarado, W. F. Tinney, and M. K. Enns. Sparsity in large-scale network computation. In C. T. Leonde, editor, *Advances in Electric Power and Energy Conversion System Dynamics and Control - Part 1*, volume 41 of *Control and Dynamic Systems*, pages 207–272. Academic Press, 1991.

[CS95] J. V. Camarda and M. A. Stadtherr. Frontal solvers for process simulation - local row ordering strategies. AIChE Fall Meeting, Miami, November 1995.

[DD93] T. A Davis and I. S. Duff. An unsymmetric multifrontal method for sparse LU factorization. Technical Report TR-93-018, Computer and Information Sciences Department, University of Florida, Gainesville, Florida, March 1993.

[DER89] I. S. Duff, A. M Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1989.

[DGL92] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection, release I. Technical report, Rutherford Appleton Laboratory, October 1992.

[DR93] I. S. Duff and J. K. Reid. MA48, a fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical report, Rutherford Appleton Laboratory, October 1993.

[Duf77] I. S. Duff. MA28 - a set of fortran subroutines for sparse unsymmetric linear equations. Report r8730, AERE, HMSO, London, 1977.

[Geo73] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[Geo74] A. George. On block elimination for sparse linear systems. *SIAM J. Numer. Anal.*, 11(3):585–603, June 1974.

[GL80] A. George and W. H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Transactions on Mathematical Software*, 6:337–358, 1980.

[Haj80] I. N. Hajj. Sparsity considerations in network solution by tearing. *IEEE Transactions on Circuits and Systems*, CAS-27(5):357–366, May 1980.

[HK73] J. E. Hopcroft and R. M. Karp. An nsup5/2 algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HR72] E. Hellerman and D. C. Rarick. The partitioned preassigned pivot procedure (p4). In D. J. Rose and R. A. Willoughby, editors, *Sparse matrices and their applications*, pages 67–76. Plenum Press, New York, 1972.

[Ior90] M. Iordache. On the analysis of large-scale circuits. *Bulletin Scientific Electrical Engineering*, 52(2):71–80, 1990.

[Kro63] G. Kron. *The Piecewise Solution of Large Scale Systems*. London, England: MacDonald, 1963.

[LM77] T. D. Lin and R. S. H. Mah. Hierarchical partition - a new optimal pivoting algorithm. *Mathematical Programming*, 12:260–278, 1977.

[Mor89] S. Moriyama. Large scale circuit simulation. *Transactions of the IEICE*, E72(12):1326–1335, December 1989.

[MS95] J. U. Mallya and M. A. Stadtherr. A new multifrontal solver for process simulation on parallel/vector supercomputers. AIChE Fall Meeting, Miami, November 1995.

[NNA90] M. Nishigaki, T. Nobuyuki, and H. Asai. Hierarchical decomposition for circuit simulation by direct method. *Transactions of the IEICE*, E73(12):1948–1956, December 1990.

[NNA91] M. Nishigaki, T. Nobuyuki, and H. Asai. Availability of hierarchical node tearing for mos circuits. *Transactions of the IEICE (Japan)*, J74A(8):1176–9, August 1991. in Japanese.

[SVLK77] A. Sangiovanni-Vincentelli and Chen Li-Kuan. An efficient heuristic cluster algorithm for tearing large-scale networks. *IEEE Trans. Circuits and Systems*, CAS-24(12):709–717, December 1977.

[SW64] R. W. H. Sargent and A. W. Westerberg. Speed-up in chemical engineering design. *Trans. Inst. Chem. Engrgs.*, 42:190–197, 1964.

[SW84a] M. A. Stadtherr and E. S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting I, reordering phase. *Computers and Chemical Engineering*, 8(1):9–18, 1984.

[SW84b] M. A. Stadtherr and E. S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting II, numerical phase. *Computers and Chemical Engineering*, 8(1):19–33, 1984.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[TV96] I. B. Tjoa and D. K. Varvarezos. Significance of problem structure in chemical process optimization strategies. *SIAM Journal of Numerical Analysis*, 1996. to appear.

[Vla85] M. Vlach. LU decomposition and forward and backward substitution of recursively bordered block diagonal matrices. *IEEE Proceedings*, 132 Pt.G(1):24–31, February 1985.

[WHMW79] A. W. Westerberg, H. P. Hutchinson, R. L. Motard, and P. Winter. *Process Flowsheeting*. Cambridge University Press, 1979.

[Woo82] E. S. Wood. *Two-Pass Strategies of Sparse Matrix Computations In Chemical Process Flowsheeting Problems*. PhD thesis, University of Illinois at Urbana-Champaign, January 1982.

[ZMDS] S. E. Zitney, J. Mallya, T.A. Davis, and M. A. Stadtherr. Multifrontal techniques for chemical process simulation on supercomputers.

[ZS94] A. I. Zecevic and D. D. Siljak. Balanced decompositions of sparse systems for multilevel parallel processing. *IEEE Trans. Circuits and Systems, I: Fundamental Theory and Applications*, 41(3):220–232, March 1994.

# CHAPTER 6 TEARING ALGORITHMS

## 6.1 ABSTRACT

New algorithms are presented to partition a sparse matrix that improve matrix reorderings and subsequent factorizations. These algorithms make use of a model hierarchy in the form of an ASCEND instance DAG but may be applied to any system of equations grouped hierarchically. The complexity of the partitioning algorithm is shown to be $\log(n) \cdot (O(n) + O(\tau))$. The algorithm is tested on a number of a large matrices and is shown to give faster reorderings that give much lower fill, much reduced operation count and faster factorizations.

## 6.2 INTRODUCTION

Graph partitioning is an important problem that has extensive applications in many domains. These include matrix reordering, VLSI design and task scheduling [KK95c]. The problem of partitioning means different things for different domains. For example, the solution of a sparse system of linear equations **Ax = b** by parallel iterative methods requires the ability to do fast matrix-vector multiplies. To do this quickly requires consideration of minimizing communication between processors and is a graph partitioning problem associated with the matrix **A**. For chemical engineering flowsheeting problems using the *sequential modular* method of solution, the problem is to partition the network graph associated with the flowsheeting units, such that a calculational sequence can be determined.

For workers in parallel computing [KK95c], the graph partitioning problem is often defined as

> partition the graph into p roughly equal parts, such that the number of edges connecting vertices in different parts is minimized.

Zecevic et al. [ZS94] consider the partitioning of a graph associated with a sparse matrix **A** so as to obtain a Bordered Block Diagonal (**BBD**) structure, as this is considered to be a desirable matrix structure for factorization. They mention that a number of algorithms based on diverse concepts, ranging from node clustering and tearing to various forms of graph dissection, have been employed to achieve these forms. For workers in sequential modular solution techniques, a block lower triangular (**BLT**) form is desirable. Though the multitude of meanings, the graph partitioning problem may be defined most generally as

> *finding a* ***good*** *set of vertices and edges so that when they are removed from the graph a new graph is obtained that possesses properties which are better than the original graph*

It is established that the search for a *good* or *best* set of edges and vertices makes the graph partitioning problem NP-complete. For this work it is necessary to define the properties that are sought from a partitioned graph.

In Chapter 5 it was argued that, for efficient solution of a linear system of equations **Ax = b**, it was sometimes advantageous to put the system in a **BLT** form first. It was also argued that *within* the non-triangular diagonal blocks arising from this form, that a **BBD** form is desirable. This work then seeks a graph partitioning algorithm which will

> take an irreducible matrix (or fully connected graph) and partition it such that a **BBD** form results where there are no *induced blocks* that are greater than some threshold size $n_{max}$, while not making the border too large.

Though this requirement may sound very similar to what most workers desire from a **BBD** form, it is different in the following way. A full factorization of the resulting matrix will be done and not a block factorization. The partitioning is primarily to find regions of a sparse matrix where good reorderings can be obtained. As such there is not the fairly stringent requirement of keeping the border as narrow as possible. This is consistent with the work presented in Chapter 5.

The rest of this chapter is organized as follows:

- a discussion of the different structures that arise from a structured modeling language.
- a presentation of the new hierarchical partitioning and reordering algorithms (TEAR_DROP) and a discussion of their complexity and performance.
- a comparison of the similarities and differences to previous work.
- numerical results.

Finally a discussion is done of the possible extensions and the implications that the algorithm might have on the application programming interface (API) for sparse matrix packages.

## 6.3 TREES AND DAGS

### 6.3.1 BACKGROUND

Trees are very common structures. A directed acyclic graph (DAG) is a specialization of a tree. Many problems are naturally represented hierarchically, i.e., as trees. This yields a notion of precedence or, as in the case of structured modeling, a part-whole notion. The ASCEND modeling language supports this part-whole notion through its `MODEL` construct. The ASCEND language also supports the notion of *aliasing* through its `ARE_THE_SAME` construct. This results in a situation where an instance of a model or atom has more than one parent. An *instance* of an ASCEND model is then a DAG, where each model constitutes a supernode. The term *supernode* and model will be used interchangeably.

.

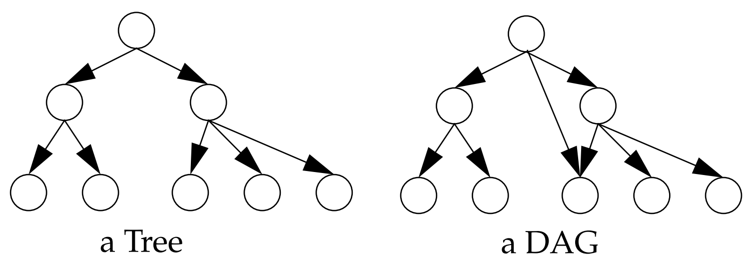



FIGURE 21 Trees and Directed Acyclic Graphs (DAGs)

Within each ASCEND model new variables and equations may be introduced. At the same time it is possible to introduce new relations in a model *without* introducing any new variables; the variables in the subtree of a model are visible to the model and are said to be within its scope. Relations (rows) can belong to only one model, but variables (columns) may be shared between models. Because the tree is directed a model lower down the tree (towards the leaves) cannot refer to the variables introduced in models above it. The variables and relations introduced in the model will later become the columns and rows of a sparse matrix.

There are a number of different ways of traversing trees and DAGs. DAGs require special treatment to ensure that a node is not visited more than once. Of the infinite ways of traversing a tree, there are a number of structured traversals. These include breadth first and depth first. This may be combined with bottom up and top down traversals.

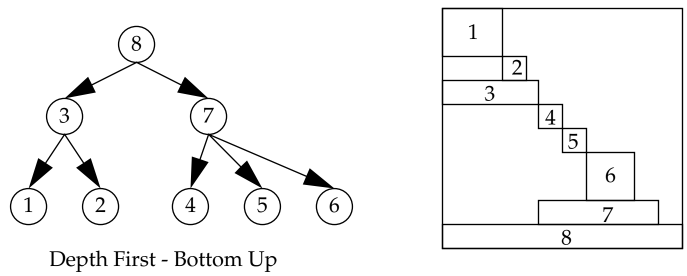



FIGURE 22 A model tree and its associated sparse matrix traversed Depth-First, Bottom-Up

Different traversal of *trees* give rise to different numbering of the nodes as shown in Figure 22 and Figure 23. If the relations and variables associated with the models in a tree or dag are numbered *while* numbering the models, different *natural* sparsity patterns will be obtained. It is these natural sparsity patterns that may be exploited to motivate different partitioning algorithms. In the figures shown, the numbered areas represent regions of the matrix where elements can potentially occur. The unnumbered areas represent regions where elements can *never* occur.

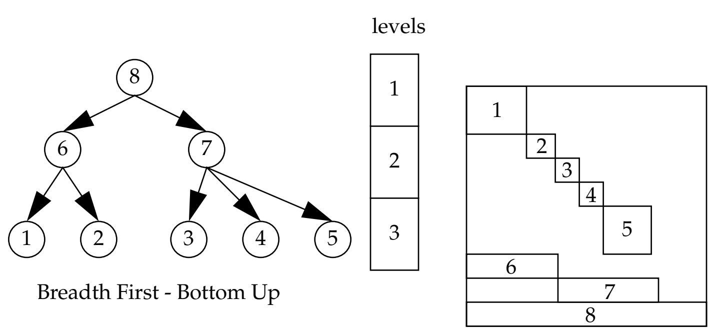



FIGURE 23 A model tree and its associated sparse matrix traversed Breadth-First, Bottom-Up

The situation is similar for ASCEND DAGs but becomes complicated by the aliasing of variables; the prediction of the location of the nonzero elements is not as well defined as with trees. This will affect the efficiency of some partitioning algorithms.

### 6.3.2 PARTITIONING

By visualizing the different natural sparse structures that are obtained from different traversal sequences, it is possible to arrive at algorithms that will yield **BBD** or **RBBD** (recursive bordered block diagonal) structures.

- Using depth first traversals, select groups of models and *cluster up* the tree or *divide down* the tree.
- Using breadth first traversals, *cleave* groups of models while working down the tree (from root to leaves).

The reason why the desired structures are obtained is explained through the following small example.

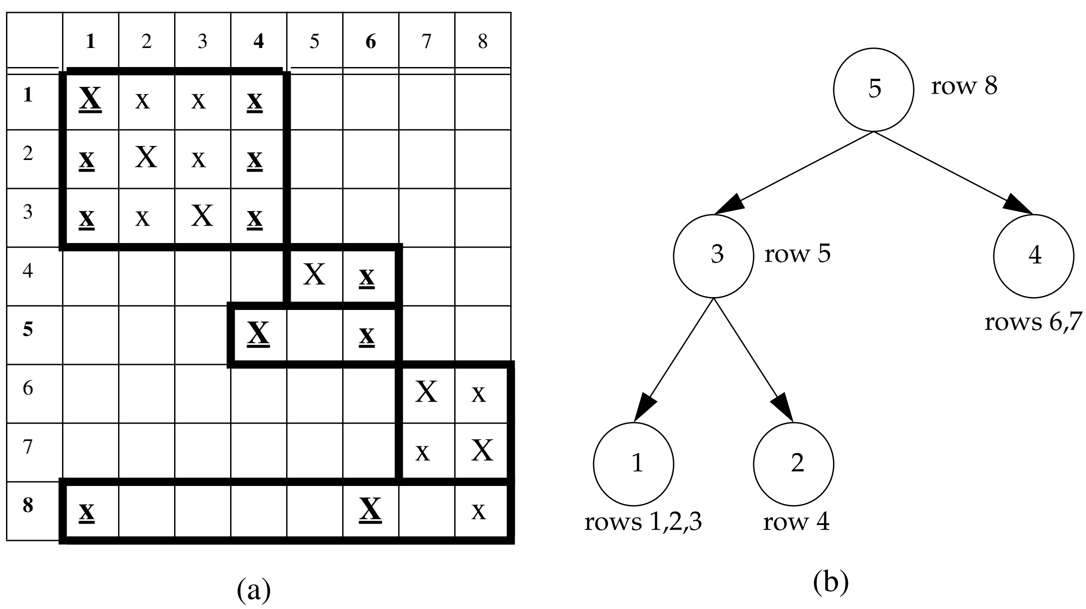



FIGURE 24 An incidence matrix and its corresponding tree.

The matrix in shown in Figure 24a represents the natural sparsity pattern obtained from a depth-first numbering of a model tree (Figure 24b). Both *x* and *X* represent incidence, while *X* represents the particular column assigned to the given row. If the matrix were to be divided into upper and lower halves (partitions) such that rows 1 - 4, are in the first half and rows 5 - 8 are in the second half, an examination of the incidence pattern shows that columns 1, 4 and 6 share variables between the two partitions, (the shared variables are shown underlined). If these columns are then removed, along with their matching rows, (1, 5 and 8) then the 2 partitions will become disjoined. This is shown in Figure 25.

In this trivial example, the region **A11** is fortuitously decoupled. If it were not, then the process would be repeated, with the rows 2, 3 and 4 forming one partition and the rows 6 and 7 forming another partition.

A11

| | 2 | 3 | 5 | 7 | 8 | 1 | 4 | 6 |
|---|---|---|---|---|---|---|---|---|
| 2 | X | x | | | | **x** | **x** | |
| 3 | x | X | | | | **x** | **x** | |
| 4 | | | X | | | | | **x** |
| 6 | | | | X | x | | | |
| 7 | | | | x | X | | | |
| 1 | **x** | **x** | | | | **X** | **x** | |
| 5 | | | | | | | **X** | **x** |
| 8 | | | | | **x** | **x** | | **X** |

FIGURE 25 The partitioned matrix

Algorithms for performing these partitionings are described in the next section.

## 6.4 ALGORITHMS

### 6.4.1 DEPTH-FIRST PARTITIONING

The algorithm that will be described uses a bisection of a model DAG. It operates on DAG that has had it supernodes numbered in a depth-first bottom-up manner, and at completion will provide a matrix which is in **BBD** form. The basic algorithm will first be described. The modifications necessary to obtain a **RBBD** matrix will then be described. The data structures required to achieve efficiency will then be examined. The algorithm is called TEAR_DROP.[1]

### 6.4.2 PARTITIONING FOR A BBD MATRIX

1. Visit the model dag with a depth-first bottom-up traversal numbering the models, relations and variables, and assembling a model list, a relations list and a variable list. Create an empty master list in which to store pointers for the tear variables.
2. Construct the sparse matrix associated with the relation and variable list.
3. Output assign the matrix; if a maximum transversal cannot be obtained, **quit**; the matrix is structurally singular.
4. Choose a cutoff size $n_{max}$.
5. Perform a block lower triangular permutation of the region **A**.
6. For each diagonal block $k$,
   1. If blocksize($k$) < $n_{max}$ **continue**.
   2. Bisect the block at a model boundary, such that the number of rows in the first set of models is almost equal to those in the second set of models. Each set of rows constitutes a partition. Label the rows in the first partition. The models that contain these rows are now *active*.
   3. Find all columns in the block that intersect both partitions. These become the tear columns. Move these columns and their matching rows out to the lower right hand corner of the block, to create a border **A12**, **A21**, and **A22**, and a main region **A11**.
   4. Add the tear columns to the master tear list and update the count of tears found.
   5. Reset the active models, and their relation counts. Reset the relation partition counters. Set **A** = **A11**. **goto** 5.
7. **quit**.

This completes the partitioning phase. To reorder the matrix, assemble all the sub

1. TEAR_DROP for tearing, decomposition, reordering and partitioning.

regions **A22** into a single border **A22** and perform a block lower triangular permutation of the resulting region **A11**. The region **A11** will now contain blocks which are guaranteed to be less than order $2n_{max}$ unless a model existed which was greater than size $n_{max}$. Reorder **A11** using an appropriate sparsity preserving reordering. If there are super-diagonal elements in the region **A22**, then reorder this region as well.

**Steps 1 - 3** constitute an initialization step and setting up some data structures to ensure efficient processing. The only step in this phase which incurs an additional cost over a basic reordering routine is the labelling of the relations with the index of the model to which they belong and initializing some counters. This additional cost is then $O(n)$ .

```
struct slv_model {
  int local;           /* the count of local relations */
  int active;          /* flag to tell if active in block */
  struct gl_list_t *relations;/* list of relations */
};

struct slv_relation {
  int model;           /* the container model */
  int partition;       /* the partition of the relation */
};
```

FIGURE 26 Basic Data Structures

The output assignment step which must be done in any case to establish the structural rank of the problem (by establishing a zero free diagonal) is not charged to the algorithm.

**Step 4** requires the choice of an $n_{max}$, which is the maximum blocksize that the algorithm should allow. This is heuristic and will depend upon the sparsity preserving reordering that will eventually be applied to the induced blocks.

**Step 5** is part of the main recursive loop and is used to find all fully connected components of the remaining matrix. The block lower triangular permutation, if done by Tarjan's algorithm is $O(n) + O(\tau)$ , *provided* that a structurally zero free diagonal exists. The other steps of the algorithm *do* maintain a zero free diagonal by ensuring that only symmetric permutations are performed. As it is being called recursively, each time with a submatrix which is at worst case, half the size of the preceding matrix, the cost is

$$(O(n) + O(\tau)) + 2\left(O\left(\frac{n}{2}\right) + O\left(\frac{\tau}{2}\right)\right) + 4\left(O\left(\frac{n}{4}\right) + O\left(\frac{\tau}{4}\right)\right) + \ldots + 2^m\left(O\left(\frac{n}{2^m}\right) + O\left(\frac{\tau}{2^m}\right)\right) \quad \textbf{(39)}$$

which is $O(n) + O(\tau)$ . A bound on the number of bisections, *k*, will be established in the next sections.

**Step 6.2** is the heart of the routine and does the bisectioning. It uses the routines **mark_relations_in_block** and **accumulate_counts**.

#### 6.4.2.1 mark_relations_in_block

1. Given a square region of the matrix, set markers *high* = -1, *low* = infinity.
2. For each row in the block,
    1. Find its original row index.
    2. Obtain the model index *k* associated with the original row, and set the active flag of model *k* to TRUE. Increment the count of local relations in model *k*.
    3. Set *high = max(high,k).*
    4. Set *low = min(low,k).*
3. **quit**.

This procedure tells which models are active in the block, sets up the count of relations that are active (some may have already been partitioned out), and establishes an upper and lower bound on where to do processing of the model tree. This step is crucial, for otherwise a search would have to be made of the *entire* model tree. As the blocks become smaller there will be fewer and fewer models in a given block, and thus one can use the markers high and low to avoid wasted searching as well be shown momentarily.

If *high* = *low* at the end of this stage, then a block has been found that *cannot* be torn by this algorithm because the philosophy that tearing should not take place within model boundaries is respected. This happens in the case where a model exists whose size is greater than $n_{max}$ such that all the rows in the block belong to the single model. Any graph partitioning algorithm [KK95c], [HL93] could then be applied to tear this *oversize* model.

#### 6.4.2.2 accumulate_counts

This routine uses the values of *high* and *low*, obtained from **mark_relations_in_block**.

1. Set *stop* = -1; set *accum = 0*.
2. If *low* = *high*, unmark the models and continue with the next block or invoke a non-hierarchical partitioning algorithm.
3. For each model *k*, $low \le k \le high$,
    1. If model *k* is not active, continue.
    2. *accum* = *accum* + *models.count*;
    3. Mark all relations for model *k* as being in partition 1.
    4. If *accum* **<** $n_{max}$ **continue,** else *stop* = *k*; **break**.
4. **quit**.

This completes the bisectioning phase. A bound on the complexity of this phase may be established by considering the following:

It takes *log k* steps to partition a graph G into *k* parts using bisection [KK95a], thus this loop will be called at most *log k* times where *k* may be estimated as $n/n_{max}$.The cost of determining whether a model is active is $O(1)$. The cost of the loop then is dominated by labelling the relations of which there may be at most *n*, (and decreasing by half with each partitioning), requiring on average *n/2* labellings, leading to $O(n)$ complexity for this code.

**Step 6.3** requires finding the columns that intersect more than 1 partition and requires at worst case examining every element of the matrix and is hence $O(\tau)$.

**Step 6.4** simply involves resetting the counts of the active models and their associated relations and hence runs in $O(n)$.

The overall complexity of the partitioning phase of the TEAR_DROP algorithm is then

$$\log(n) \cdot (O(n) + O(\tau)) \tag{40}$$

### 6.4.3 PARTITIONING FOR A RBBD MATRIX

Another algorithm, which is very similar to the basic TEAR_DROP algorithm, is one which provides an **RBBD** structure. In the *reordering* step of the TEAR_DROP algorithm, a single border is constructed by symmetrically permuting all the tear columns and their matching variables to the lower right hand corner of the matrix. The region **A11** is then subjected to a final block lower triangular permutation and each nontrivial diagonal block is reordered. If, however, the partitioning and reordering steps are combined, i.e., each time an induced block falls below its threshold size $n_{max}$, it is immediately reordered, then an **RBBD** structure will be obtained. In this case Step 6.4 of the basic TEAR_DROP algorithm is omitted.

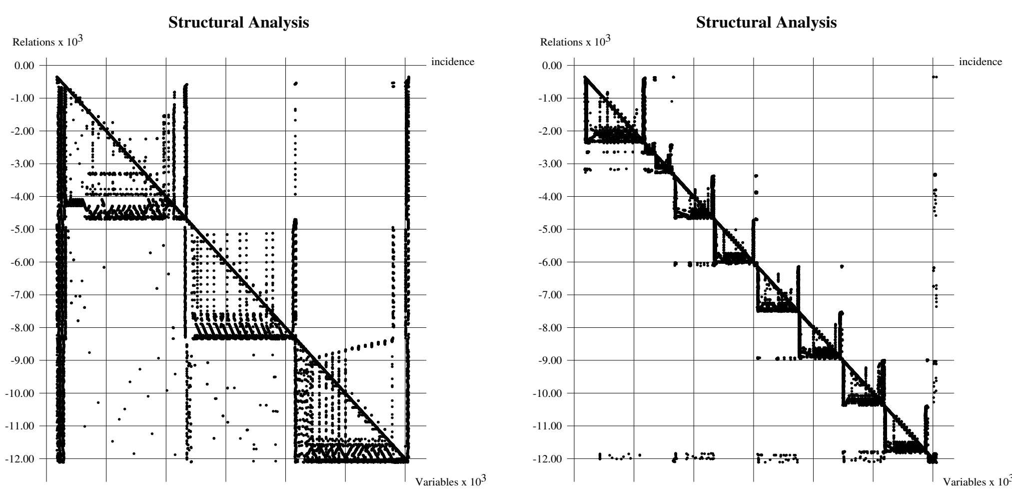



FIGURE 27 Standard Reorder versus TEAR_DROP algorithm

Figure 27 shows a typical reordering when using the **RBBD** TEAR_DROP algorithm. The matrix in this example was the 4Cols matrix described in Chapter 5.

### 6.4.4 BREADTH-FIRST ALGORITHMS

A potentially very simple algorithm for obtaining a **BBD** structure is based on breadth first traversals of a tree. In Figure 28 (a) for example, if all the relations in model 8 are removed, then the problem breaks into 2 disjoint parts, containing models 6, 1 and 2 and models 7, 4 and 5 respectively. Repeating the process would yield the situation as shown in Figure 28 (c).

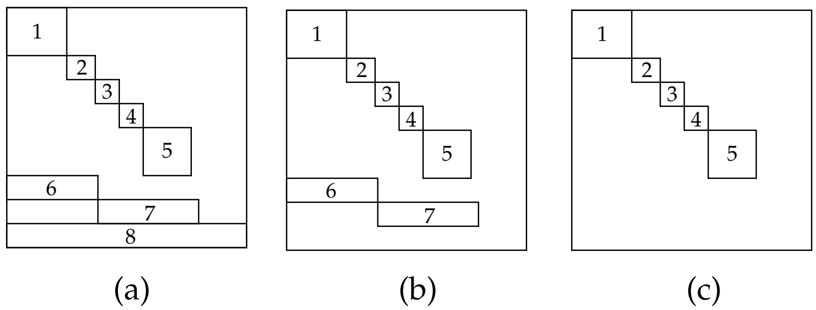



FIGURE 28 Breadth-First Cleaving

A prototypical algorithm would then be

1. Visit the model DAG with a breadth-first bottom-up traversal numbering the models, relations and variables, and assembling a model list, a relations list and a variable list.
2. Construct the sparse matrix associated with the relation and variable list.
3. Output assign the matrix; if a maximum transversal cannot be obtained, **quit**; the matrix is structurally singular.
4. Set **A11** = **A**.
5. Select the model with highest index and find all rows belonging to the model. Find the matching columns for these rows.
6. Symmetrically permute these rows and their columns to the lower right hand corner of the matrix to form a border. Update the region **A11**.
7. Perform a block lower triangular permutation of the region **A11**.
8. Foreach induced block larger than $n_{max}$ **goto** 5.
9. **quit**.

The reordering phase would then reorder the resulting non-trivial diagonal blocks using a suitable sparsity preserving reordering algorithm.

This breadth-first algorithm was not fully investigated, as it can fail if the model tree is really a DAG, which is the most common situation with the ASCEND system. This is illustrated in the figure below, where models 1 and 3 have more than one parent, and significant cleaving would be required to decouple the problem.

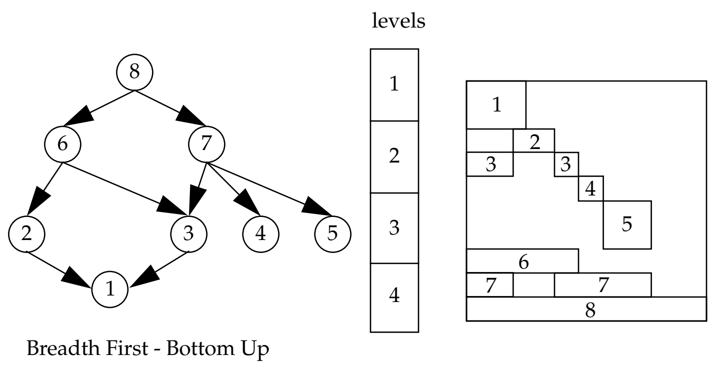



FIGURE 29 Breath-First cleaving of a DAG

### 6.4.5 CHEMICAL ENGINEERING FLOWSHEETS

Chemical engineering plants are often designed using flowsheets. A flowsheet is a flat representation of the connectivity of the units in the plant. In this form flowsheets may be thought of as being a simple 2 level hierarchy; all the connection streams comprise level 1 and the units comprise level 2. The sequential modular approach to solving flowsheeting problems involves choosing a set of tear streams that, when they are removed, yield an acyclic calculational sequence. Values are guessed for the tear streams, the flowsheet is solved in the calculational sequence found, resulting in new values for the guessed streams and possibly some sensitivity information. The guessed values are updated using some criterion and the process is repeated until the guessed values are within some acceptable bound of their computed values.

The question of choosing the tear streams is a graph partitioning problem and has been studied extensively by a number of workers [BM72], [UG75]. A feature of this problem is that a minimal set of tear streams is desirable, as it reduces the number of streams that must be iterated upon to solve the original problem.

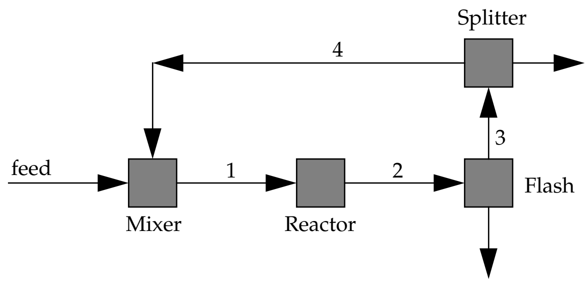



FIGURE 30 Simple recycle flowsheet.

For example in Figure 30 (assuming that there are no computational controllers present), tearing any one of the streams 1 through 4 will allow a calculational sequence to be found. An equation based solution strategy does not require a minimal set of tear streams, and indeed need not be concerned with the notion of a stream. Consistent with the earlier discussion is the need to find a set of rows and columns that will allow the problem to be decomposed temporarily for reordering.

If the problem is considered to be a 2 level hierarchy, the resulting structure would be as shown in Figure 31.

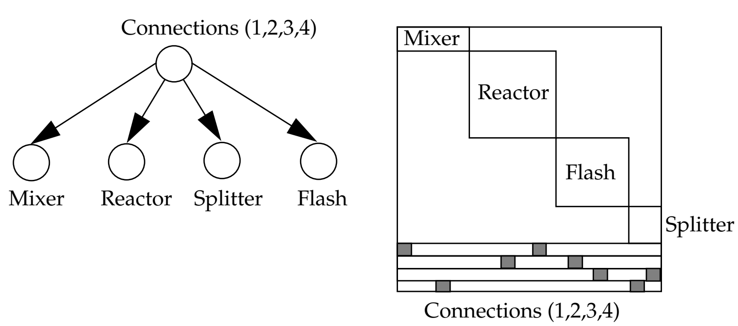



FIGURE 31 Pseudo 2 Level Hierarchy

The TEAR_DROP algorithm would find all the connections in exactly one bisection. However, this would most likely lead to very large and very small induced blocks, as some units such as a splitter have very few *internal* variables. The internal variables of a unit are variables that are not shared by any other unit. A breadth-first algorithm based on the algorithm outline presented, should perform well on this class of problems. It should be noted that in the early stages of this work an algorithm which used explicit labelling of the connection equations was developed. The results were very encouraging and led to the development of the TEAR_DROP algorithm, which is much more general in scope.

## 6.5 RELATED WORK

The notion of hierarchical partitioning or tearing is not new. What sets this work apart from the techniques of other workers is its generality and its domain independence. This work has features of Asai and Tanaka, [NNA90], who considered the use of a simulation input language to create an **RBBD** matrix. They considered only electrical engineering problems and had a structured 4 level hierarchy involving very specific circuit components at each level of the hierarchy. They also required the user labelling of input and output flags on each circuit element; nodes with both flags labelled would become tear nodes. They provide theoretical estimates of the number of fill but only tested the scheme on problems with a small number of variables.

Stadtherr and Wood, [SW84] developed their BLOKS algorithm for chemical engineering flowsheeting matrices. It may be considered as an *implicit tearing* scheme (see Section 5.5.3). It has similarity to the algorithms presented here in that it restricts reordering to certain regions of the matrix, but without explicit removal of rows or columns of the matrix. In BLOKS these regions are those associated with the flowsheeting units. The algorithm does not try to ensure irreducibility of each region, nor does it require regions to be disjoint (column wise). Reordering must then be restricted to row reordering only, or the potential disturbance of the reorderings of an already processed unit must be accepted.

Recently, Tjoa and Varvarezos [TV96] suggested that by restricting the dynamic Markowitz reordering to the rows and columns which define the units of chemical engineering flowsheeting matrices may lower the over fill in. Unfortunately they do not provide details of their algorithm nor details of its performance.

Lin and Mah, [LM77] presented a hierarchical partitioning algorithm (or rather a sequence of them), which selects a spike row and spike column, deletes them, and checks for the reducibility of the remaining system. The process is repeated until the remaining system is lower triangular. They report very good fill in properties, though note that the running time of the algorithm was very expensive. This has been confirmed by others including Duff [DER89] and Wood [Woo82]. None of these workers provide a bound on the cost of their algorithms.

Duff [DER89] Chapter 8 looks at reordering sparse matrices to special forms. He mentions that some of these special forms are designed not necessarily to minimize fill but to contain fill to certain regions of the matrix. He comments that '*although this often results in more fill-in, it is not necessarily the case, since the local strategies do not minimize the fill-in globally*'. Among the algorithms reviewed by Duff is nested dissection [Geo73]. The central concept of nested dissection is the removal of a set of nodes from the graph of a *symmetric* matrix that leaves the resulting graph in 2 or more disconnected parts. The parts are then further divided by the removal of sets of nodes, with the dissection nested to any depth. Previously George had proposed a single level dissection, which creates a 2 level tree, very analogous to the pseudo-hierarchy that may be created from flowsheeting problems. Duff in his review comments that the '*... fundamental reason for the success of the nested dissection is (that) it is truly a global ordering in the sense that decisions made at the very first stage take the entire matrix into account*.' The TEAR_DROP algorithm is thus very similar to nested dissection from the point of view of global decision making but has different objectives for doing its tearing. Additionally, no assumptions about symmetry are necessary. One of the difficulties with nested dissection is the question of what nodes should be removed in the initial dissection. Although automatic algorithms have been proposed for finding these initial nodes, this writer argues that this nested dissection is being applied to the wrong problem; it should be applied to the original model of the problem where these questions can be answered using the most information available. Application of TEAR_DROP to any hierarchical representation of a problem makes this node selection question trivial. The work of Bjørstad [Bjø95], in finite element substructuring runs very close to the philosophy of this work. In yet another interpretation, the TEAR_DROP algorithm may be thought of as employing a heavy clique matching (HCM) in the graph growing stage of Karypis and Kumars [KK95b] graph partitioning algorithm.

Finally this author while concurring with Duff and Reid that '*we need to rethink our sparse matrix algorithms*', thinks that it may be more important to '*rethink our modeling practices*'.

## 6.6 NUMERICAL RESULTS

In Chapter 5, the performance of a number of different sparse codes that make use of better *a priori* reorderings was examined. In those tests the TEAR_DROP algorithm used in conjunction with the **SPK1** reordering was seen to perform well. In this section, some more details concerning the performance of the TEAR_DROP algorithm itself are given.

The scope of these tests is limited. In particular, only one sparsity preserving reordering (SPR) was used. The results here are best interpreted as the basis for determining reasonable values $n_{max}$ for one solver/SPR combination. They can also be used to evaluate in more detail the reduction in reordering times presented in Chapter 5. The data reported there were the *total* partitioning and reordering times, and the breakdown is presented in the following tables. The partitioning times can be used to evaluate the claimed complexity of the TEAR_DROP algorithm.

The **LU1SOL** solver was used for all the tests wherever factorization times are reported. The test matrices used were those presented in Chapter 5. For each matrix tested the following information is provided

- the partitioning time with TEAR_DROP.
- the total partitioning and reordering (**SPK1**) time.
- the number of induced blocks (equal to the number of reorderings done).
- the size of the largest induced block.
- the number of tear columns found.
- the factorization time.

**Table 22: Statistics with RBBD TEAR_DROP**

| Matrix and order/# models | $n_{max}$ | Partition Time | Total Partition and Reorder | # of induced blocks | Largest induced block | Number of Tears | Factor Time |
|---|---|---|---|---|---|---|---|
| Isom_30K | | | | | | | |
| 19995/14201 | 1000 | 4.97 | 5.22 | 356 | 995 | 2592 | 0.85 |
| | 2000 | 4.56 | 5.33 | 346 | 1995 | 2532 | 0.86 |
| | infinity | 0 | 27.73 | 0 | 19995 | 0 | 16.45 |
| 4Cols | | | | | | | |
| 12456/1482 | 1000 | 1.66 | 2.49 | 22 | 993 | 1330 | 2.85 |
| | 2000 | 0.87 | 2.86 | 9 | 1997 | 364 | 3.08 |
| | infinity | 0 | 20.42 | 0 | 11770 | 0 | 22.82 |
| 10Cols | | | | | | | |
| 31140/3702 | 1000 | 7.48 | 10.03 | 53 | 998 | 3081 | 7.33 |
| | 2000 | 4.26 | 9.44 | 26 | 1965 | 1040 | 7.62 |
| | infinity | 0 | 118.62 | 0 | 29496 | 0 | 115.55 |
| BigEquil | | | | | | | |
| 8986/2545 | 1000 | 0.27 | 0.48 | 17 | 855 | 479 | 0.52 |
| | 2000 | 0.19 | 0.65 | 9 | 1788 | 302 | 0.93 |
| | infinity | 0 | 1.91 | 0 | 3961 | 0 | 4.72 |
| PPP | | | | | | | |
| 14698/5780 | 1000 | 1.17 | 1.90 | 44 | 986 | 323 | 0.35 |

**Table 22: Statistics with RBBD TEAR_DROP**

| Matrix and order/# models | $n_{max}$ | Partition Time | Total Partition and Reorder | # of induced blocks | Largest induced block | Number of Tears | Factor Time |
|---|---|---|---|---|---|---|---|
| | 2000 | 0.64 | 2.21 | 16 | 1992 | 111 | 0.38 |
| | infinity | 0 | 10.27 | 0 | 8201 | 0 | 0.41 |
| Wood7 | | | | | | | |
| 6858/2284 | 1000 | 0.16 | 0.38 | 37 | 891 | 144 | 0.19 |
| | 2000 | 0.11 | 0.54 | 21 | 1677 | 57 | 0.19 |
| | infinity | 0 | 1.50 | 0 | 3508 | 0 | 0.24 |
| Wood8 | | | | | | | |
| 17509/4444 | 1000 | 1.49 | 2.17 | 19 | 948 | 1017 | 6.51 |
| | 2000 | 1.09 | 2.42 | 16 | 1876 | 744 | 4.36 |
| | infinity | 0 | 13.51 | 0 | 9087 | 0 | 26.47 |
| Ethyl_60 | | | | | | | |
| 59080/31907 | 1000 | 6.90 | 8.44 | 434 | 995 | 1872 | 2.31 |
| | 2000 | 3.86 | 7.0 | 358 | 1998 | 1272 | 2.92 |
| | infinity | 0 | 50.67 | 0 | 20316 | 0 | 13.22 |
| Ethyl_80 | | | | | | | |
| 79554/44800 | 1000 | 20.76 | 22.84 | 631 | 997 | 3000 | 3.96 |
| | 2000 | 11.66 | 15.99 | 563 | 1966 | 2164 | 3.94 |
| | infinity | 0 | 204.94 | 0 | 50172 | 0 | 32.34 |

## 6.7 DISCUSSION

A number of new algorithms for partitioning a sparse matrix have been presented. These algorithms exploit the structures inherent in a hierarchical representation of a model. One of these algorithms, TEAR_DROP, which is based on recursive bisectioning was described in some detail and the complexity of $\log(n) \cdot (O(n) + O(\tau))$ was proved. On the problems tested this algorithm was seen to be reasonably fast, while providing significant improvements in the overall solution efficiency for the **LU1SOL** code. These results should extend to any code that can take advantage of better *a priori* reorderings.

There are a number of possible extensions to the work that has been presented. In particular, the breadth first algorithms based on node cleaving should be investigated further. As mentioned previously, sparse matrix reorderings other than **SPK1** should be investigated. It is expected that other reordering algorithms will have different optimal values for $n_{max}$. A value of 1000 was found to be satisfactory throughout this work.

The implementations of the reordering algorithms and block lower triangular permutation algorithms used in this study are by no means optimal. For $n_{max}$ = 1000, an average of 83% of the analysis time is spent in the partitioning algorithm with the balance being spent in the reordering algorithm. The implementation of the block lower triangular permutation code needs to be re-examined. Similarly, the implementation of the reordering algorithms need to be re-thought in the context of being called many times on problems no greater than twice the $n_{max}$ value.

The algorithm presented is naturally parallelizable. Each induced block may be submitted to a different processor on multi-processor architectures because they are disjoint; tearing and reordering the left partition is independent of the right partition.

The output assignment algorithm is a potential bottleneck to large scale linear algebra. It's complexity is $O(n\tau)$ and has to be done at the moment over the entire problem. Although in practice it does not normally exhibit its worst case behavior, Ben Allan (private communication) has found a number of random matrices of moderate order, with row counts that are a multiple of 3, that result in very long run times for the code output assignment code, **mc21a**. It is of interest to see whether an algorithm based on bottom up clustering of a model dag could not be used to break the $n^2$ effect inherit in this algorithm. Equally as interesting is the question 'a*re some matchings better than others?*'. Since a maximal matching is not unique, the effect of the matching on the eventual performance of the tearing and reordering algorithms is an open question.

The TEAR_DROP algorithm does not try to be minimal in the number of tears. Intuitively there should be a limit on how much tearing can be accommodated before inferior factorization results are seen. A slight modification to the algorithm (in the tearing step) can be used to put a limit on how many tears will be allowed at each partitioning stage. This was implemented with a limit that no more than 10% of the size of any given block may be torn. The limit was only reached with a matrix called petro16K. The use of such a limit also changes the complexity bound of the algorithm.

The TEAR_DROP algorithm performed poorly on the petro16K matrix. This matrix is a chemical engineering flowsheeting matrix and is planar. The rows and columns corresponding to the flowsheeting units was known as well as the connection equations. As predicted in Section 6.4.5, the TEAR_DROP algorithm found all the connections in the first bisection, which resulted in the problem being overtorn; the subsequent factorization was worse than the untorn problem.

Finally the efficiency of the algorithms presented needs to be examined on a much wider class of problems.

## 6.8 APPENDIX

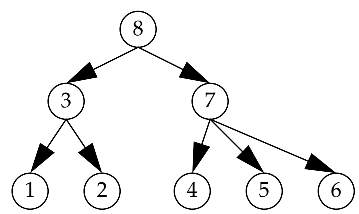
### 6.8.1 MATRIX FORMATS

In order to make use of use of the hierarchical structure of a system, it is necessary to be able to communicate this information to a sparse matrix package or to a preprocessor to such a package. Most sparse matrix packages require that their data be presented in some defined way. Of the many sparse matrix formats that have developed (Saad [Saa94] describes *15* different formats), two popular ones have emerged and have become *defacto* standards. These are the Harwell-

Boeing format [DGL92], and the SMMS format [Alv83].

The algorithms presented here require a model-row file which, apart from the statistics of the problem (the number of super-nodes or models and the number of rows), has a list of model-row tuples. This implicitly assumes that the information was collected in a bottom-up depth first manner. However, for general communication of a problem the traversal scheme from which this indexing was derived has to be given explicitly. The connectivity information of the supernodes should be provided and a graph format not unlike that used in the Metis [KK95b] system or the dot package, [Kou93] should be used. An example is seen in Figure 32. The grammar is simple; each line represents a node index and the index of its children. Leaf nodes are stated implicitly.

```
digraph {
  3 -> 1,2;
  7 -> 4,5,6;
  8 -> 3,7;
}
```




Depth First - Bottom Up

FIGURE 32 Model connectivity information

Thus if the following information is communicated, any other queries can be derived (for example a model-column relationship can be had from a knowledge of the sparse matrix pattern and model-row information).

- sparse matrix row, column information and numeric values in some format (Compressed Row or Triplet format for example).
- supernode graph connectivity.
- traversal order, DF/BF, BU/TD.
- supernode-row file tuples.

## 6.9 REFERENCES


[Alv83] F. L. Alvarado. The sparse matrix manipulation system, May 1983.

[Bjø95] P. Bjørstad. Large scale direct solution of finite element equations. volume ILAY workshop on Direct Methods. CERFACS, September 1995.

[BM72] R. W. Barkley and R. L. Motard. Decomposition of nets. *Chem. Eng. J.*, 3(265), 1972.

[DER89] I. S. Duff, A. M Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1989.

[DGL92] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection, release I. Technical report, Rutherford Appleton Laboratory, October 1992.

[Geo73] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[HL93] B. Hendrickson and R. Leland. The chaco users guide, version 1.0. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[KK95a] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, University of Minnesota, Department of Computer Science, June 1995.

[KK95b] G. Karypis and V. Kumar. Metis, Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, Department of Computer Science, August 1995.

[KK95c] G. Karypis and V. Kumar. Multilevel *k*-way partitioning scheme for irregular graphs. Technical Report 95-064, University of Minnesota, Department of Computer Science, August 1995.

[Kou93] E. Koutsofios. *Drawing graphs with dot; dot User's Manual*. AT&T Bell Laboratories, Murray Hill NJ, October 1993.

[LM77] T. D. Lin and R. S. H. Mah. Hierarchical partition - a new optimal pivoting algorithm. *Mathematical Programming*, 12:260–278, 1977.

[NNA90] M. Nishigaki, T. Nobuyuki, and H. Asai. Hierarchical decomposition for circuit simulation by direct method. *Transactions of the IEICE*, E73(12):1948–1956, December 1990.

[Saa94] Y. Saad. *SPARSKIT: a basic tool kit for sparse matrix computations*. University of Minnesota, June 1994.

[SW84] M. A. Stadtherr and E. S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting I, reordering phase. *Computers and Chemical Engineering*, 8(1):9–18, 1984.

[TV96] I. B. Tjoa and D. K. Varvarezos. Significance of problem structure in chemical process optimization strategies. *SIAM Journal of Numerical Analysis*, 1996. to appear.

[UG75] R. S. Updadhye and E. A. Grens. Selection of decompositions for process simulation. *AIChE J.*, 21(136), 1975.

[Woo82] E. S. Wood. *Two-Pass Strategies of Sparse Matrix Computations In Chemical Process Flowsheeting Problems*. PhD thesis, University of Illinois at Urbana-Champaign, January 1982.

[ZS94] A. I. Zecevic and D. D. Siljak. Balanced decompositions of sparse systems for multilevel parallel processing. *IEEE Trans. Circuits and Systems, I: Fundamental Theory and Applications*, 41(3):220–232, March 1994.

# CHAPTER 7 CONCLUSIONS

This thesis has examined the issues associated with Very Large Scale Modeling (VLSM). The hypothesis that a dedicated environment is necessary for efficiently conducting large scale modeling has been proved.

The fundamental features of a VLSM environment have been presented. A language, ASCEND IV, to support such an environment has been proposed and a prototype language and environment, ASCEND IV.alpha, has been implemented. In addition, algorithms that exploit the structures associated with large scale problems have been developed which can improve the linear algebra computations associated with solving systems of equations. Throughout this report the important results of this work have been summarized and directions for future work have been suggested.

The results of this work suggest that it will take 200 seconds to create a problem with 100,000 variables and that it will require 150 MB of memory for representation. 40 seconds will be required to do a function and gradient evaluation and that an unsymmetric matrix factorization of the sparse Jacobian matrix associated with the problem will require less than 10 seconds. In less than 20 minutes it will be possible to instantiate and to solve a square nonlinear system with 100,000 variables from scratch.[1]

However, modeling is a much more complicated process than solving well understood problems. The VLSM environment proposed and presented in this work satisfies the needs of exploratory and evolutionary design. In so doing, a modeler can efficiently conduct his or her modeling task from conceptual design through final design. With appropriate provisions for sharing of data through persistent representations, the entire design process involving many modelers may be accommodated.

Perhaps the greatest impediment to very large scale modeling is the post analysis of the results of the modeling exercise: *what does one do with a vector of 250,000 values?* This question is not unrelated to the information hiding and information filtering issues raised in Chapter 3 and the need for modeling support tools discussed in Chapter 2. Unfortunately, this work, though recognizing these problems has done very little to alleviate them.

Finally, the algorithms and assumptions used in this work should be sufficient for problems with up to 1,000,000 variables. The next natural goal of 10 million variables, or *ultra large scale modeling,* will build upon the experiences of this work and the work of others who are looking at design in the large.

1. assuming 10 Newton iterations.