HYDROMETALLURGICAL SIMULATION - A VIABLE PROGRAM STRUCTURE

Johannes Jacobus le Roux Cilliers

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering

Johannesburg, 1986

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

J.J. Cilliers

31 day of MARCH 1987

ABSTRACT

In order to satisfy the increasingly sophisticated requirements for simulation in the metallurgical industry, this research focused on the development of data structures and algorithms general and extendable enough to accomodate all hydrometallurgical processes.

The data structures designed to describe process stream information were based on the concept of *substreams* (Britt, 1980) using the plex data structure (Evans, Joseph and Seider, 1977) inherent to the Pascal programming language. As substreams are combined to describe complete process streams, the data structures may be extended to describe any process stream by creating additional substreams as may be required.

Algorithms for partitioning, tearing and ordering flowsheets based on the work of Tarjan (1972, 1973) and Lee and Rudd (1966) were designed and implemented. These algorithms are able to treat the large problem sizes associated with hydrometallurgical process flowsheets.

The data structures and algorithms have been successfully combined into a powerful process simulator extendable to the general hydrometallurgical process description.

CONTENTS Page

DECLARATION ii
ABSTRACT iii
CONTENTS iv
LIST OF FIGURES vii
LIST OF TABLES viii

1 INTRODUCTION 1

1.1 *The Use of Steady-state Simulation in* Metallurgy 1
1.2 Simulation Strategies 2
1.2.1 The sequential modular approach 3
1.2.2 Equation oriented methods 3
1.2.3 Two-tier algorithms 4
1.2.4 Comparison of methods 4
1.3 The Requirements of an Advanced Process Simulator 6
1.4 An Evaluation of Existing Metallurgical Simulators 8
1.5 Comparison of Ore-dressing and Hydrometallurgical Simulator Structures 11
1.5.1 Comparison of ore-dressing and hydrometallurgical stream structures 11
1.5.2 Comparison of ore-dressing and hydrometallurgical process operations 13
1.6 Conclusion 14

2 SIMULATOR DATA STRUCTURES 15

2.1 The Plex Data Structure 15
2.2 Achieving the Plex 16
2.2.1 Simulating the plex in FORTRAN 16
2.2.2 The plex as it exists in other languages 17
2.3 The Choice of Programming Language 18
2.3.1 Overview of languages 18
2.3.2 Language choice 19
2.4 The Stream Descriptive Data Structure 22
2.4.1 Introduction 22
2.4.2 The stream types 25
2.4.3 The solid substream 26
2.4.4 The complete stream structure 30
2.5 The Unit Operation Structure 35
2.5.1 Introduction 35
2.5.2 The unit data structure 35
2.6 Data Structure Usage 37
2.6.1 Introduction 37
2.6.2 Utility routines 38
2.7 Conclusion 42

3 STREAM TYPE CHARACTERISATION 43

3.1 Introduction 43
3.2 Stream Types and Computer Memory 44
3.3 Stream Types and Convergence 47
3.4 Stream Type Allocation 51
3.4.1 User specified stream types 51
3.4.2 System types 52
3.4.3 Computer allocated types 53
3.4.4 The choice of methods 54

4	PRECALCULATION ALGORITHMS	56
4.1	Introduction	56
4.2	Partitioning the Flowsheet	57
4.3	Tearing the Flowsheet	59
4.4	Determination of the Stream-cycle Matrix	66
4.5	Determination of the Calculation Order	69
4.6	Verification	69
4.7	Conclusion	70
5	CONCLUSIONS	73
5.1	Introduduction	73
5.2	Summary	73
5.2.1	Data structures	73
5.2.2	Precalculation algorithms	74
5.2.3	Verification	75
5.3	Future Research	76
5.3.1	Introduction	76
5.3.2	Unit operation models	76
5.3.3	Convergence	80
5.3.4	Human-computer interfaces	81
5.4	Conclusion	81
REFERENCES		82

LIST OF FIGURES

Figure		Page
2.1	Schematic of solid substream	28
2.2	Schematic representation of a solid-lixiviant stream	33
2.3	The unit operation data structure	36
2.4	Schematic of data structure indicating pointer name	39
3.1	Typical uranium flowsheet	46
3.2	Hypothetical process	49
4.1	Example flowsheets for tearing and ordering	71
5.1	Flowsheets for system verification	77,78

LIST OF TABLES

Table		Page
4.1	Verification of modified Lee and Rudd algorithm	72
5.1	Results of simulations	79

CHAPTER 1 : INTRODUCTION

1.1 The Use of Steady-State Simulation in Metallurgy.

The research performed dealt exclusively with steady-state simulation techniques, and any reference to *simulation hereafter shall* imply steady-state simulation.

The use of simulation in the metallurgical industry has shown a tremendous growth in the past five years. Not only have simulators moved from the university and consulting environment into industry, but simulation has come to be recognised as a viable engineering tool.

Simulators have, in the past, been used as powerful predictive units on a large scale. This included usage as design tools, for the prediction of alternate circuit configurations and for the optimisation of circuit operating parameters. Should simulators be more freely available, their usage may very well include operator training, the prediction of simple plant *parameter changes, and* the familiarisation of metallurgists with the variations in plant operation with variations of operational parameters.
The need for simulators of high quality that are easy to use and reliable, and that will instill confidence in users unfamiliar with simulation, is therefore obvious.

The use of simulation in South Africa has, in the past, been restricted mainly to the ore dressing side of mineral processing, for two reasons. Firstly, the modelling of ore dressing operations is relatively simple, and many models based on historic data or on empirical fitting exist. Secondly, available simulators have not been able to accommodate the complex data structures that exist in hydrometallurgy, both in terms of the data for each stream in the process flowsheet, and the complex structures of the flowsheet itself. The design of this simulation program was aimed at solving the second of the above problems. The modelling of hydrometallurgical unit operations is in itself a large task, and research in this field is active.

Thus the aims of this project were to produce a simulator that would be able to operate on a small computer, thereby making it accessible to a large number of users, while simultaneously laying down the program structures required so that the simulation of hydrometallurgical circuits is also made possible. This program shell, capable of accepting hydrometallurgical unit models and their associated stream structures, will further stimulate research in the modelling of these unit operations, as the tedium of designing the associated code for the tearing of recycles, the ordering of the calculation phase and the design of data structures has been eliminated.

1.2 Simulation strategies.

The primary mathematical problem in steady-state process simulation is one of solving large systems of nonlinear algebraic equations. There are three specialised approaches to solving these equations:

the sequential modular approach, the equation-oriented approach and two-tier algorithms.

1.2.1 The sequential modular approach.

Essentially all industrial simulators available at present operate on the sequential modular approach (Evans, 1980). A computer routine is developed for each unit operation to produce the output stream variables as functions of the input stream variables and unit parameters. Each routine is then sequentially called to simulate the complete process. Initial estimates of the recycle stream variables must be provided, either by default or by the user, the unit calculation order being sequentially repeated until convergence of the recycle streams to within a specified tolerance has been achieved. The solution of design problems or constrained problems is also done by iteration. Before a simulation can be done, a host of precalculation algorithms have to be called to determine the calculation order for each routine and the streams that have to be assumed for the simulation to be started.

1.2.2 Equation oriented methods.

These methods collect all of the equations describing the flowsheet and solve them as a large system of nonlinear algebraic equations. In general, the system of equations is sparse and structured. In this approach, the models generate and represent the equations for each unit operation. These equations are then fed to an efficient equation solving procedure. Using this method a simulation may be formulated as an optimisation or design problem. The equation oriented method has been used extensively in models for individual unit operations, but have not

been used routinely in an industrial simulator (Evans, 1980).

1.2.3 Two-tier Algorithms.

Using this approach, the overall solution strategy of the sequential-modular approach is maintained, while both rigorous and simple models are made available. The simple models are in the form of a group of linear equations that can be rapidly solved by an efficient equation solving technique to determine all the stream variables. This allows the rigorous models to be called when a more accurate simulation is required. This method has been used in an advanced simulator. (Evans, 1980). Other two-tier algorithms include that of Westerberg et al (1979), known as the simultaneous-modular approach, where all the stream variables are solved simultaneously using simple linear models. The two-tier approach takes advantage of all the existing sequential modular software. This approach may be considered as another convergence method, where simple, fast models are used to rapidly approach the solution; the final values being determined by rigorous modelling.

1.2.4 Comparison of methods.

Writing the executive for an equation-solving simulator can become extremely complex (Westerberg et al, 1979). The problems are those of guaranteeing that the dressing engineer will give a legitimate problem definition to the system and that the solution can be made to converge to a solution. The sequential modular and two-tier approaches are much simpler to write. The assumption that each unit operation model will calculate unit output stream values given input stream values and equipment

been used routinely in an industrial simulator (Evans, 1980).

1.2.3 Two-tier Algorithms.

Using this approach, the overall solution strategy of the sequential-modular approach is maintained, while both rigorous and simple models are made available. The simple models are in the form of a group of linear equations that can be rapidly solved by an efficient equation solving technique to determine all the stream variables. This allows the rigorous models to be called when a more accurate simulation is required. This method has been used in an advanced simulator. (Evans, 1980). Other two-tier algorithms include that of Westerberg et al (1979), known as the simultaneous-modular approach, where all the stream variables are solved simultaneously using simple linear models. The two-tier approach takes advantage of all the existing sequential modular software. This approach may be considered as another convergence method, where simple, fast models are used to rapidly approach the solution; the final values being determined by rigorous modelling.

1.2.4 Comparison of methods.

Writing the executive for an equation-solving simulator can become extremely complex (Westerberg et al, 1979). The problems are those of guaranteeing that the dressing engineer will give a legitimate problem definition to the system and that the solution can be made to converge to a solution. The sequential modular and two-tier approaches are much simpler to write. The assumption that each unit operation model will calculate unit output stream values given input stream values and equipment

parameters solves many problems. By assuming a rigid form for writing a unit model, these routines may be written comparatively easily as the writer has to make no decisions on the type of input data that he will be given. In the equation-solving approach, the modeller has to manipulate the equations defining the unit into a form recognisable by the system.

The executive for the modular approaches has to analyse the flowsheet and determine a computation order for the unit models. Methods for solving this problem are well established.

For the equation solving approach two options exist: The equations may be solved by successive linearisation or by an analysis attempting to minimise the number of iterate variables required. Using the former method, the system must calculate or estimate partial derivatives and set up a set of simultaneous linear equations for each iteration. The latter alternative can be very complex and represents a massive combinatorial problem as the use and position of each equation must be determined. The complex form has not been used for simulators (Westerberg et al, 1979).

By not using the equation-solving technique, the optimisation or design problem, requiring equipment parameters to be calculated, becomes more complex and greatly increases the computation time.

Modular simulation executives may be designed and written more quickly, the system is simple to specify and unit models may be written, tested and made computationally robust. The user further has no difficulty in specifying data to produce a well-defined problem. All of the magnitude of

presently available modular unit models may further directly used if this approach is adopted.

The sequential modular method was used for the present simulator on the basis of the above arguments. Experience with the MODSIM simulator as successfully implemented by Ford and King (1984) further indicated the suitability of this approach to the present work. The simulator was further written in such a way that the two-tier approach as described by Evans (1984) may be implemented at a later stage should the robust models prove to be too slow computationally for optimisation or design usage.

Henceforth, reference to a "simulator" or "simulation" shall imply a steady-state, sequential modular sytem.

1.3 The Requirements of an Advanced Process Simulator

As early as 1976 (Evans and Seider), it was recognised that the direction which simulation techniques must take had to be clearly defined. The problems of simulations containing solids in the process streams, and the adequate representation thereof were recognised and the requirements of such a simulator set.

These requirements, if applied to the steady-state simulation of hydrometallurgical plants, are as follows:

1. The system must permit analyses of flowsheets with different types of streams.

2. The system must be extendable and capable of modification.

3.The system must be adaptable to different computing environments, and transportable.

4.The cost of the simulator and performing simulations should be reasonable.

5.The system should be file oriented,the results of one phase being stored before analysis of the next.

6.The system must be conveniently accessible to the user.

7.The assumptions employed in an analysis should be clear.

8.The user should be [illegible] to communicate in a convenient form, and it should be easy to interpret results.

9.The system should be easy to learn to use and well documented.

Only the first six of these apply directly to the research done. Although the last points are by no means less important, they are the result of good human-computer interface design and of the models used in the simulation, and are beyond the scope of this work. The executive program designed essentially operates from and writes to data files for the interface to manage. The use of such a modular program structure allows the interchanging of different sections to suit the requirements of the ultimate user.

The first requirements of portability, accessibility and cost point to the use of small, or preferably micro computers. In the time since Evans and Seider set those requirements, computer power has increased exponentially, with a simultaneous decrease in cost and size. It is now possible for each user to have a dedicated machine, and the software we create must face this reality. The future of computers is clearly in the direction of smaller machines, with a proportional increase in their power and speed. The use of local computing power conforms to the predictions of Evans (1980) for the simulator environment of the 1990's.

The second major requirement is for a general and extendable simulator that will be able to process any size problem, with a wealth of different stream types and unit operations. This work was primarily concerned with the design of the data structures and algorithms that would meet this second requirement, while not losing sight of the first.

1.4 An Evaluation of Existing Metallurgical Simulators.

A literature survey of metallurgical simulators capable of simulating hydrometallurgical flowsheets or which will be extendable to represent the complex data associated therewith satisfactorily, was undertaken. No simulator exists at present that meets the requirements for simulating hydrometallurgical operations, although the concepts employed for the representation of process stream data and general data structures are well established.

The extension of an existing simulator to meet advanced requirements has been done by Ritchie and Spencer (1984), with great success. The ASPEN-PLUS system produced as an extension of ASPEN, an advanced chemical process simulator (Evans et al. 1979), is howver extremely large, consisting of approximately 350 000 FORTRAN statements, and is not implementable on a micro-computer. The extensions made by Ritchie (1984), while allowing for economic evaluation of minerals processing operations and greatly enhancing the ASPEN system for use in metallurgical simulations, does not yet allow adequate data structures for the representation of complex solids that are to be represented in multiple dimensions of characterisation. The ASPEN-PLUS system allows solids to be distributed according to size and mineral content (or grade) and the latter to be distributed according to specific gravity. The system does not allow true multi-dimensional distributions, e.g. a specific gravity distribution in each size class. Further dimensions for distribution, for example flotation rate constants and unreacted core size is also not catered for. ASPEN does however use very elegant data structures for the representation of streams data (Britt, 1980) and state-of-the-art algorithms for the precalculation phase, during which the flowsheet is analysed and the calculation order determined (Evans, 1980). The bulk of the code size and unfamiliarity with the code make extensive modifications to the program by unfamiliar users difficult.

The Monsanto FLOWTRAN system uses data structures similar to ASPEN, however only simple extensions have been attempted (Neville and Seider, 1980). Once again, the system is written in FORTRAN, with large sections of code being used for data manipulation and

memory allocation and management. The difficulties associated with maintaining code written to simulate a high-level language, as was done in both ASPEN and FLOWTRAN, become evident from their work.

Hess and Wiseman (1984) describes the ore-dressing simulator of the Julius Kruttschnitt Mineral Research Centre. Although the human-computer interface is elegant, the system is main-frame bound and limited to ore-dressing operations. In concept it is similar to MODSIM, the ore dressing simulator of the University of the Witwatersrand (Ford and King, 1984). Extensions to the latter appeared feasible, however the memory allocation of the program is done by recompilation of the main FORTRAN executive and linking of the associated code - a very time-consuming task on a small computer. The rigid FORTRAN array based data structures further complicate extensions to allow the multiple solid and lixiviant phase descriptions required for hydrometallurgical simulations. The MODSIM simulator is limited to a true three-dimensional matrix type solid phase description.

The FLEXMET system of Fluor (Richarson et al, 1980 and 1981) cannot describe the solid phase in any great detail and is therefore unsuitable for use.

The literature survey clearly indicated that for the steady-state simulation of hydrometallurgical operations, preferably on a micro-computer, no suitable, extendable programs exist and that a simulator executive would have to be designed and created combining concepts of other simulators to meet the advanced requirements.

1.5 Comparison of ore-dressing and hydrometallurgical simulator structures.

The hydrometallurgical simulator should be seen as an ore-dressing simulator extended to accommodate lixiviant data rather than an extension of a chemical process simulator to accommodate solids data. The adequate representation of the large amount of solids data associated with minerals processing operations is one of the major problems in data structure design for a simulator; a problem investigated and effectively solved by Ford (1976). A comparison of hydrometallurgical and ore-dressing simulators rather than chemical simulators should be made before the conceptual extensions can be done.

1.5.1 Comparison of Ore-dressing and Hydrometallurgical Stream Structures

Consider the simulation of ore-dressing circuits as compared to the problem of hydrometallurgical, or in general, any metallurgical process.

The process streams that make up an ore-dressing circuit consist of a combination of solids and water, the solids being classified into different fractions. Depending on the complexity of the simulation, these fractions or classes are generally up to three dimensions deep, being size, grade and one further characteristic. Grade in this context refers to the percentage of valuable mineral in the particle. The last characteristic may be any further physical or chemical attribute, such as magnetic susceptibility, flotability or shape.

This representation is conveniently available by the usage of a matrix-type data structure, each characteristic being represented by a dimension of the matrix. The fraction of the total solids tonnage in each class is stored in the corresponding position in the matrix, from where it is easily accessible and understandable. In general, all programming languages has this data structure available. The simulator MODSIM implements the above representation very successfully in FORTRAN.

The structure required of a hydrometallurgical process stream is more complex as the additional information required for lixiviant streams and their associated solutes is often required. Information regarding the core size, porosity and shape of solid particles may be required and further increases the amount of data to be represented. The further possibility of gaseous stream types, and the associated thermodynamic qualities of *hydrometallurgical process streams*, should these be required, must further be considered.

Hydrometallurgical process models, also beyond the scope of this work, are being developed on a population balance approach (Sepulveda and Herbst, 1978). As little information is available as to the exact nature of these models and the parameters and data that they shall require, the data structure must, of necessity, be variable and maintainable. By the same token, since no knowledge is available before a simulation is performed about the amount of the data that will be required, the data structure must be able to expand or contract to suit the needs of the specific system under consideration.

1.5.2 Comparisons of Ore-dressing and Hydrometallurgical Process Operations

The simulation of systems of ore-dressing operations leads to a few generalisations that cannot be made when simulating the general minerals processing operation. The most obvious of these is the restriction of a single feed to all unit operations, except for mixers, where only a single product stream is allowed. Although this at first appears to be a rather innocuous limit, the simplifications that result in the algorithms that precede the actual simulation phase; partitioning, tearing and precedence ordering, is considerable. These precalculation algorithms are discussed in detail in Chapter 4. In the general hydrometallurgical circuit, this restriction is clearly not possible - we may cite a liquid-liquid extraction unit as an obvious example of a unit operation to which the single feed or single product restriction cannot be applied.

A lesser restriction, that all process streams in the circuit shall be of the same type, leads to simplification in the generalisation of the code produced - this becomes a severe problem when considering hydrometallurgical simulations, as shall become obvious in later sections.

A further simplification in the simulation of ore-dressing circuits results from the small number of recycles that occur. Recycle streams in the analysis of any complex flowsheet requires that the steady-state simulation be solved by an iterative process. All the recycles in the process have to be isolated, streams that constitute a part of the cycle have to be opened or torn, their initial values guessed and iterations performed to convergence. In

general, a large ore-dressing operation shall have *fewer than* five recycles in any one connected section, and larger numbers shall usually only occur in systems where much recycling of water is done. The latter occurs in coal washing operations rather than mineral processing plants.

In hydrometallurgical operations, on the other hand, multiple recycle systems frequently occur, and in a system of a few units with multiple feeds and products, in the order of tens of recycle loops may occur. This large number of recycles influences both the precalculation algorithms and the calculation phase of the simulation, and algorithms that were perfectly adequate for the ore-dressing simulator fall short when hydrometallurgical systems are attempted.

1.6 Conclusion

The simulation of the general metallurgical process is a complex matter, and simulators that up to date have been restricted to ore-dressing are not adequate. The general hydrometallurgical process simulator requires the representation of different types of process streams, a maintainable and variable data structure and robust and general precalculation algorithms.

The design of these data structures and the algorithms that will make the simulation of hydrometallurgical circuits possible were the two aims of the research performed.

CHAPTER 2: SIMULATOR DATA STRUCTURES

2.1 The Plex Data Structure

In the early stages of the design of the simulator, it was already recognised that rigid data structures, as prescribed by certain programming languages such as FORTRAN and BASIC, would not be adequate for the elegant representation of either a general process stream, or a general process simulator. This is primarily due to the greatly varying nature of simulation problems, both in terms of possible problem sizes and the stream types to be represented.

Evans, Joseph and Seider (1977) proposed a "plex" data structure that met the requirements of their advanced process simulator. The plex consists of groups of contiguous storage locations known as beads; the number of locations in each bead being variable. A bead may contain different types of entries, such as integers, real values, booleans or alphanumeric strings.

A bead is referenced to by a pointer, which refers to the storage location of the entire bead. A bead may also contain a pointer to the next bead, so that beads may be strung together.

Beads are created dynamically during execution of a program from a pool of free storage, and may be returned to that pool when no longer required.

Evans and Seider claim that the plex structure will:

1. Increase modularity.
2. Provide flexibility.
3. Allow a more natural description of attributes.
4. Make configuration modifications simple.
5. Not waste storage.
6. Not place a limit on problem size.

The implementation of the plex data structure therefore allows the creation of a general process simulator, able to accomodate a vast range of simulation problems.

2.2 Achieving the plex.

There are two ways in which the plex may be achieved; the code to simulate the plex structure may be written for programming languages that do not inherently have the structure (such as FORTRAN), or a programming language that supports the plex structure may be used.

2.2.1 Simulating the plex in FORTRAN.

A plex structure, written in FORTRAN was used by Evans et al (1979) for the ASPEN chemical process simulator. This simulator was extended by Ritchie (1984) to include solids in a simulation, as ASPEN-PLUS. This simulator consists of more than 350000 lines of FORTRAN code. Kaijaluoto (1979) also created a plex data structure for their simulator and describes the problems with using FORTRAN. He finds that the maintaining and administration of the plex requires large amounts of storage space and that the creation of a data structure that will not lead to programming errors by the user is both difficult and time-consuming. The plex created is also slow and complicated to use.

The ASPEN simulator uses its own input language that may be seen as a new programming language required to be known to effect a simulation.

The creation of the code to simulate the plex is not simple and becomes a major time factor in the creation of new simulator. The maintainability and transportability of such a simulator is also reduced by the additional specialised code.

2.2.2 The plex as it exists in other languages.

The dynamic allocation of computer memory to suit the requirements of the problem, and the accessing of the data segments created by means of a pointer is available in all the Algol family of languages. This family includes Pascal and C.

When using a language that contains the plex as an inherent data structure, programming emphasis is shifted from creating the code that maintains the data structure to using the available structures to their full potential. More effort may therefore be put into designing and using the best algorithms available, both for the simulation and precalculation phase.

A major disadvantage when not using FORTRAN is the loss of familiarity and the required learning of a new language if the code is to be updated by a user unfamiliar with the language. Algol-type languages are however well-defined and their use is to be preferred to the situation where an existing language is adapted to meet advanced requirements for which it was not designed.

2.3 The Choice of a Programming Language.

2.3.1 Overview of languages

FORTRAN and BASIC, while not inherently allowing the plex, have further undermined their maintainability by the constant addition of new features, such as dynamic arrays in BASIC, and certain structured programming constructs in FORTRAN 77, which were not present in earlier versions. These changes lead to severe transportability problems, as each computer inevitably supports its preferred version. While BASIC has become a standard language for micro-computer applications, it is not suitable for the development of large programs.

The new generation of programming languages holds great advantages for the user. Data structures have become less rigid, and unique structures applicable to the problem that is being solved may be set up by the user. These include the dynamically created plex as *described above. The use of recursive subroutine* calls has become an acceptable programming technique, leading to more compact code, and more elegant solutions to simple problems. The dynamic allocation of memory is possible, with obvious benefits for the *problem size that may be solved, and the creation of* programs that will solve the general rather than a specific problem.

While a very large selection of languages are *available, very few* of these have found general acceptability, especially in the scientific and engineering communities. Those considered included Pascal, C, Ada, Simula and some even lesser known. For micro-computers, Ada and Simula may be excluded,

the compiler for the former being so large that it cannot be used, and for the latter generally unavailable. C is becoming a greatly favoured language. It has an exceptionally fast execution rate, compilers are available, and the language produces very compact code. It is however a very low level language, meaning that is closer to machine code than languages such as BASIC or FORTRAN, and many standard features available in other languages have to be created by the user. Apparently debugging C programs is also no simple matter and the maintainability of programs is low; the code being complex.

2.3.2 Language choice.

It became evident that Pascal was becoming a much favoured language for general applications. Compilers are freely available, and are not only fast, but relatively cheap. Large amounts of application software is available, making the creation of programs to do graphics, mathematical manipulations, or data capture quite simple. Pascal is further being used as a language to teach programming to student engineers at some universities, making the future acceptability of the language more certain.

Pascal has all of the features inherent to FORTRAN as standard, with all the qualities of the newer Algol family languages. Its applicability to the requirements of this simulator became evident when some of the features not found in FORTRAN were considered in detail:

Data structures.

Pascal encourages the use of data structures that suit the problem. It is for this reason that the plex data structure is so convenient to use, as any entity that can be described by a combination of distinctly different pieces of information may still be allocated a single data structure, this in turn consisting of the predefined combinations of descriptive information. Pascal further allows the definition of any suitable data type for the problem, which once defined becomes part of the structures inherent to the language. These structures may consist of types of variables, or ranges of values that any variable may take on, checking being performed during execution for out-of-bound values. Other inherent data types include sets, strings, arrays and arrays of arrays, or matrices. These inherent structures and the addition of defined structures make the generation of error-free code easier, and the maintaining of such code much simpler.

Dynamic storage allocation.

This feature allows problems of any size to be executed by one single program. If one section of a program grows, the memory required for that section may be allocated, while the remaining sections remain constant. This means that the maximum problem size need not be predefined, and that problems consisting of greatly variable dimensions may be effectively solved. This is essentially the implementation of the plex described by Evans, Joseph and Seider (1977).

Structured programming.

The control structures of Pascal include all those contained in the newer FORTRAN versions, but are more convenient to use by the definition of blocks of code, started with a BEGIN and terminated with an END

statement, the block then being treated as a single statement. This makes the structuring of programs much simpler, and the usage of the GOTO statement is practically eliminated.

Recursive subroutines.
The use of recursive procedures, or procedures that are allowed to call themselves from within the procedure, make many algorithms much simpler to implement. This construct is illegal in FORTRAN, and often leads to bulky and inefficient code where a recursive routine could be used. This is especially true of graph theoretical applications where depth first searches are used, as is in simulation.

Language definition.
One final point should be mentioned. Pascal is a very well-defined language, with a clearly defined standard. This means that if programs are written that do not make use of any extensions to the language, the programs should be transportable. The compilers available for microcomputers are readily available for many types of machines, and hence if a well-known compiler is used as standard, transportability from one PC to another becomes a minor problem.

Conclusion.
Pascal appears to have all the characteristics required of a language used for large system development, and was used for the complete simulator. This is not to say that the problem could not have been solved using another language, such as FORTRAN, but that the convenience of creating the code, and the maintainability of the final product required a more advanced language. The data structures developed in any large program are a function of the language

used, and by using Pascal, it is felt that problems may be solved in terms of their inherent structure, rather than *having to* be transformed into a rigid data structure definition. More time could therefore be allocated to designing good algorithms and *code*, rather than on ways to make the data conform to the language and manipulating these rigid data *structures*.

2.4 The Stream Descriptive Data Structure

2.4.1 Introduction

The representation of the complex, multiphase material flowing between *process units* in a general way has been one of the great problems of simulator design. Only the ASPEN chemical process simulator has solved the problem adequately at present. Their design of process streams has been extended by Ritchie (1984) *to include* minerals processing streams. Britt (1980) describes the ASPEN multiphase stream structure in great detail. He bases the ASPEN structure on the requirements set by Evans and Seider (1976).

Britt remarks that "in general it may be stated that any portion of a stream that *is to be treated in* a special manner by unit operation blocks need to be carried separately from the rest of the stream. These separate portions of a stream are called "substreams"". Streams may therefore be subdivided into substreams, each *representing a* portion of the stream that is to be treated diffently by unit operation models.

ASPEN recognises two types of stream components, "conventional" and "nonconventional". Conventional components (in the chemical sense) represent pure compounds or pseudo-compounds that may be characterised by standard properties such as molecular weight, critical pressure and ideal gas heat capacity coefficients.

Nonconventional components, such as coal, ash, slag wood pulp and, by extension, most ores, cannot have their thermodynamic properties calculated by conventional methods, such as equations of state. Nonconventional components are instead characterised by vectors of data representing the physical properties of the components. These physical attributes, as state variables, are carried as stream data, since they may change from stream to stream. ASPEN considers mixtures of nonconventional components as all being of a single phase.

ASPEN further allows the use of three substream types, MIXED, NC and CISOLID. A MIXED substream represents the flow of any number of phases in equilibrium. NC components are "nonconventional"; they are considered inert with respect to equilibrium calculations but enter into energy balance relations. The CISOLID substream represents the flow of "conventional" components that are considered inert with respect to phase equilibrium calculations, but not chemical equilibrium calculations.

It is clear that most substreams used for the calculation of metallurgical operations will fall into the nonconventional class, due to the presence of solids and dissolved species. The concept of substreams is however very elegant for the representation of complex process streams, and was

used for the present simulator. Substreams representing separable phases rather than conventional, nonconventional and equilibrium condition were used.

Thus any hydrometallurgical process stream may be described as the combination of a set of substreams, each of these substreams representing a seperable phase, for example solids, carbon, lixiviants or an organic phase. By describing the information pertinant to a specific substream type as a record, substreams may be combined to describe any process stream in general.

The substream descriptive structure can broadly be described as follows:
Each stream type that exists has a pointer type unique to it. A stream can be of any type when created; each stream having a record containing all possible stream pointer types. Once the storage and descriptive requirements, effectively the type, of the stream are known, that pointer that describes the stream type is initialised, bringing into existance all the required memory associated with the stream. The remaining pointers are not used. The efficiency and ease of manipulation of this pointer description far outweighs the memory wasted by having all possible stream type pointers available. The memory used for the description of a single pointer variable is small; only four bytes are used.

Consider a stream containing solids and some lixiviant, such as the product from a leach tank. Once the stream has been created for a simulation, it can take on many possible types. To fix the stream type, only the solid-lixiviant stream type pointer is initialised. The initialisation of this pointer

brings into existance the memory required to describe the solid and lixiviant substreams of the complete stream. Had the stream however contained only solid material, such as the feed to a crusher, the stream type would have been of type unmixed. A further parameter would indicate that the specific unmixed substream type is solid, and only that memory is brought into existance.

As stream data storage space can be allocated dynamically, the definition of possible stream types does not actually affect the memory allocation in the computer. Only when these types are required is their memory allocated, and then only as much as is required for the specific stream type required.

The overall data structure that resulted from this study was produced by an evolutionary process. The final structure is considered optimal in that it allows the generalization of procedures that operate on the same substreams of different stream types. Single substream records can therefore be accessed and processed by a single procedure, rather than procedures having to be written to accommodate each possible stream type in the system. This clearly simplifies the writing of models to describe unit operations.

2.4.2 The stream types

In order to uniquely characterise a process stream consisting of separable substreams, two parameters are required. The first describes which combination of substreams the stream consists of, and the second, only applicable in the case of unmixed streams (or streams consisting of only one descriptive record), to describe the type of this single record. The

example mentioned earlier illustrates this requirement.

The simulator can accomodate any number of substream types, and their combination into complete stream descriptions. The simulator at present accomodates both solid and lixiviant substreams as well as the combination of these substreams to form a *solid-lixiviant stream*. This structure further allows the definition of pure liquids such as water by considering them as lixiviants containing no dissolved species. A carbon substream has been designed and implemented for the simulation of carbon in pulp systems by other workers involved in the modelling of these systems (Stange, 1985). Substreams can only effectively be designed by those that are required to use them for their models - the requirements of maintainability and versatility of the data structures thus being of paramount importance.

2.4.3 The solid substream.

As previously mentioned, the description of the solids phase requires a far larger amount of data to be stored than lixiviant substreams. The solid substream descriptive record therefore had to be designed to optimally utilise storage, and not to descibe the maximum possible problem. The actual description of the solid phase is however quite simple, requiring only the tonnage in the substream, and then data to descibe the fractions of the tonnage in all possible classes required to characterise the solid. In ore-dressing simulators, these classes generally are combinations of three characteristics, being size fractions, a grade description and one other property; flotability and magnetic

example mentioned earlier illustrates this requirement.

The simulator can accomodate any number of substream types, and their combination into complete stream descriptions. The simulator at present accomodates both solid and lixiviant substreams as well as the combination of these substreams to form a solid-lixiviant stream. This structure further allows the definition of pure liquids such as water by considering them as lixiviants containing no dissolved species. A carbon substream has been designed and implemented for the simulation of carbon in pulp systems by other workers involved in the modelling of these systems (Stange, 1985). Substreams can only effectively be designed by those that are required to use them for their models - the requirements of maintainability and versatility of the data structures thus being of paramount importance.

2.4.3 The solid substream.

As previously mentioned, the description of the solids phase requires a far larger amount of data to be stored than lixiviant substreams. The solid substream descriptive record therefore had to be designed tc optimally utilise storage, and not to descibe the maximum possible problem. The actual description of the solid phase is however quite simple, requiring only the tonnage in the substream, and then data to descibe the fractions of the tonnage in all possible classes required to characterise the solid. In ore-dressing simulators, these classes generally are combinations of three characteristics, being size fractions, a grade description and one other property; flotability and magnetic

susceptibility being the most common. In the description of hydrometallurgical processes, other characteristics may be required, such as shape, porosity and surface quality. A large amount of data may thus possibly be required to be stored. This in turn requires a data structure with practically no limits. In order to do this, a flexible data structure must be used, that structure allowing the storage capacity to expand or decrease as required, without deliterious effect to the rest of the simulator.

The flexibility required is achieved in the following way:
Rather than defining a static matrix as the descriptive part of the solids record, an array of pointers is staticly defined. Each of these pointers can be activated to point to an array of fixed size. As the total number of classes required to describe the solid substream is known before the calculation phase is entered, the exact number of pointers required may be activated. Thus the data structure takes on the form of a practically unlimited array, values being stored in contiguous locations. The data structure for the solid substream is shown schematically in Figure 2.1.

Some wastage of space must obviously be incurred when the array pointed to by the last of the initialised pointers is not filled, however this wastage is negligable compared to situations where static memory is used. The larger the array to which each pointer points, the fewer the number of pointers that need be statically defined, and the smaller the amount of static storage wasted. In contrast, the amount of allocated array space that is unused grows as the arrays get larger.

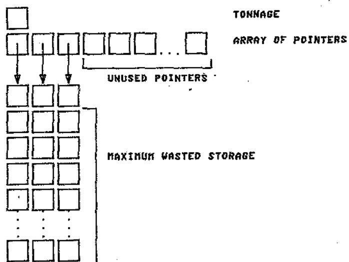


FIGURE 2.1

SCHEMATIC OF SOLID SUBSTREAM

The maximum number of variables that may be stored is therefore the product of the number of static pointers available and the length of each array of storage that it may be initialised to access. The actual dimensions of the classes used to characterise the solids are therefore immaterial, so long as the total number of variables may be accomodated. Should the maximum number of storage positions be inadequate at any stage, the number of variables that may be accomodated can be increased quite simply by increasing either the length of each dynamic array, or increasing the number of pointers that are available to be initialised. The limit on the total number of variables possible is only the limit that is set by the memory available to the user on the particular computer being used.

The accessing and storing of values in this array structure clearly requires the calculation of the positions of both the pointer and the specific position in the array it points to. To reduce the knowledge of the data structure required of users, functions were written that will perform the replacing and accessing of values from this data structure if the coordinates of the class, if considered as being part of a static matrix, are known. This makes the data structure almost transparent. These functions give the user the impression that he is dealing with a five dimensional matrix rather than a set of contiguous arrays (See also Section 2.6.2.2). Five dimensions are considered adequate for the description of any hydrometallurgical simulation, the use of all these dimensions in any one model requiring the solution of a sixth order partial differential equation (Sepulveda and Herbst, 1984). Should more dimensions

however be required to be described, the user may access all variables without the benefit of the utility routine, thus having an umlimited number of dimensions at his disposal. The maintaining of the variables in the data structure in such a case becomes the responsibility of the user.

The solids record is most comprehensively designed, and may be considered as a model for the design of other substream types, especially those that may require the storage of large amounts of data. In cases where less data is required to be stored, static storage may be used instead, the loss in efficient usage of memory space being made up for in ease of use.

2.4.4 The complete stream structure

Having described a general substream record, we are now in a position to combine substreams into complete process stream descriptions. This is done by defining records of pointers to the respective substream types. These records consist of as many pointers as we require substreams. All permutations of combinations of substreams into complete streams may be defined as stream type records. In order therefore to create a stream, or rather the memory to store the description of that stream, we simply need to initialize the record of pointers that constitutes the combination of substreams that describe the particular stream.

Each stream further requires a descriptive record to indicate its reference number, its source and destination units, and the type of stream it represents. This data may be stored in a general descriptive array, applicable to all streams. This

general array also includes information as to the exact nature of the data the stream stores, such as the number of descriptive dimensions, and the number of classes in each dimension. This makes it possible for different streams to have different descriptive records, even if they are of the same type; for example, the solid substream of a stream preceding a crusher may only require a single size class in the size dimension, while the product may require a larger number of size classes to adequately represent the material. In general though, all streams in the circuit containing a certain substream type will have the same descriptive dimensions for that substream.

Finally, there must be no limit on the number of streams in a circuit. This can be done by defining a 'large' number of static pointers, each one able to point to a stream descriptive record. When a stream is required, we initialize a pointer to create the storage space for that stream. For sequential modular simulation, the only streams required to be present at any point in the calculation phase is the feed and product streams of the unit being simulated, as well as the system feed and product streams and the tear streams in the circuit. This allows a substantial reduction in the storage requirements of any simulation, but results in an increase in the simulation time, as stream data storage space has to be allocated and disposed before any unit can be calculated. The largest time overhead however occurs during the output phase, where permanent storage and access is required on some form of disk should all the stream data not be available simultaneously. Floppy disks generally have a 320 kb storage capacity, while most PC's have a 640 kb memory capability. This memory allows at least 200 complex process streams to be kept in memory simultaneously.

It is therefore considered unlikely that a problem exeeding this size will be solved as a single simulation.

The present system therefore does not allocate and dispose stream data during the calculation phase, but keeps all stream data available in memory to facilitate better interactive data output. Should this at any time result on a limit on the realistic maximum problem size, this may be changed to a allocate and dispose system without any major reprogramming.

Consider now the complete data structure to describe stream data: (Figure 2.2)

An array of pointers to stream data structures exists. When it is required to store data pertaining to stream n, pointer n is initialised. The initialisation of this pointer results in the memory for the first level of description of the stream to be allocated. This first level of description consists of the following:

1. The flowsheet identity number of the stream.

2. A complete descriptive identity array. This array describes the stream type, the size of its class structures and its source and destination units.

3. A pointer to a next level of descriptive records. A choice of pointers exists at this level - only one pointer is initialised, that one pointing to the record having the combination of substream pointers required to mimic the true stream to be described. This pointer clearly forms the basis of the stream type, as discussed in Section 2.4.1.

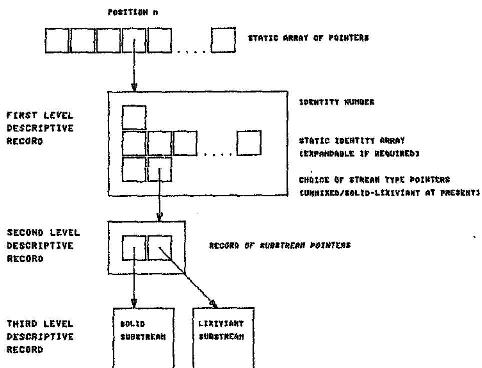


FIGURE 2.2

SCHEMATIC REPRESENTATION
OF A SOLID-LIXIVIANT STREAM

The record on the second level of description consists of as many pointers as there are substreams in the stream. These pointers in turn point to the actual data structures required for the stream. These pointers also have to be initialised to create the required memory space from the heap. Note that on this level pointers to records rather than the records themselves are used. This is primarily for the sake of the unmixed stream descriptive record, as some Pascal compilers insist on allocating the memory required for the largest of the possible choices of a variant record. By using the records themselves, the memory for the largest of the records would have been allocated, by using the pointers, no wastage is incurred.

This second level of description thus points to the actual substream records earlier discussed.

Although it may appear that an unnecessary level of description is included in the data structure in that the second level substream pointers (Figure 2.2) could have been directly available in the first descriptive level in the place of the stream type pointers, no ambiguity in the description of a stream is possible, and the maintainability of the structure is increased. New record types may be included at any of these descriptive levels to describe new stream types.

It should thus be clear that new stream types can be created in two ways, firstly by designing a pointer to a new combination of existing substream types or by designing a substream that has not existed before and combining it with other substreams to form a new stream type with a pointer to it.

2.5 The Unit Operation Structure

2.5.1 Introduction

The data structure required to store the unit model types, and the data pertaining to a specific unit in the circuit are very similar in concept to those of the streams. Once again a large array of pointers are allocated, the position of each pointer in the array corresponding to a unit reference number in the circuit. Associated with each pointer is the unit model type to which each unit in the circuit refers. Note that these are simply the unit types represented, not the actual model that is referred to. Each unit type can, of course have more than one model associated with it, and the reference code of the actual unit model that is required to be used is passed as a parameter to the unit type routines. This may be seen as a two-layered calling program, firstly the unit type that was indicated in the circuit is called, this routine in turn making the choice between the possible models that can be used for the simulation of this unit operation.

2.5.2 The unit data structure

The record that is required to be transferred to the unit model simulation procedure, is accessed by a pointer. This record contains, further to the reference code of the specific model to be accessed, a list of the number of input streams to the unit and their reference numbers, as well as the number of output streams and their reference numbers. Further the number of parameters required for the unit model and these parameters can be accessed. Figure 2.3 represents this data structure schematically.

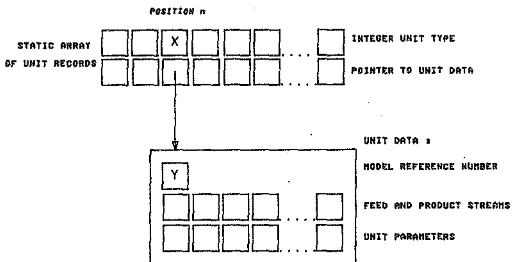


FIGURE 2.3

UNIT OPERATION DATA STRUCTURE

UNIT n IS OF TYPE X USING MODEL Y

Note that an *array of simple pointers were* not allocated, but rather an array of records, each record containing both a pointer and an integer reference code to the unit type. This reference code indicates the *unit type that is to be simulated*, for example a leach tank. This reference code is required to be available outside of the descriptive record, so that, knowing the unit reference number in the circuit *and considering that* position in the array of records of unit data, the unit type is immediately available. Thus the pointer to the remainder of the data of that particular unit operation type may be passed directly to the simulating procedure. The storage space wasted by the allocation of these unit type reference code integers is small, and is required for the effective calling of the calculation sequence.

2.6 Data Structure Usage.

2.6.1 Introduction

A certain amount of familiarity with the stream data structure is required for the writing of unit models. As the unit models for hydrometallurgy are not very well established, and the data required to be transferred from unit to unit is not yet known, the update of the data structures describing substreams must also be considered. To assist the user in writing unit models, utility procedures have been created that reduces the tedium of accessing data and updating records of unit data. The utility routines pertaining to the accessing of the solid substream are described with specific reference to Figure 2.4.

Note that the symbol "^" used to descibe the usage is Pascal for "pointing to" and is indicated diagramatically in Figure 2.4 as an arrow.

It should be clear that the exact equivalents of these utility routines may be written for any other substream type. It is recommended that *if a new* substream type is created, the equivalent routines for that substream should be written, so as to minimise the difficulties that may arise when accessing the variables of the record describing the substream.

2.6.2 Utility routines

2.6.2.1 Solidselect.
The accessing of a particular record of a substream is the basis of any unit model. Consider again the data for the solid phase, and assume that we wish to access the tonnage in a particular process stream. Let the stream number be n. The value may be changed *with:*

```
str(.n.)^.unmix^.solids^.tonnage := whatever ;
```

in the case of a stream consisting only of solids (i.e. UNMIXed) and

```
str(.n.)^.solix^.solids^.tonnage := whatever ;
```

in the case of a SOLid-LIXiviant combination stream. Note that the generic use

```
str(.n.)^.solids^.tonnage, or even str(.n.)^.tonnage
```

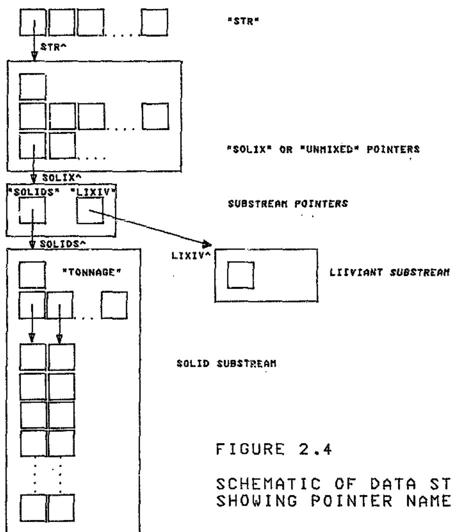


FIGURE 2.4

SCHEMATIC OF DATA STRUCTURES SHOWING POINTER NAMES

is not allowed, *even if no chance of* ambiguity exists. In order to ease this accessing, a function called SOLIDSELECT was written. This function will return the value of the pointer to the lowest level *of description. Thus we may use*

```
tempptr := solidselect(str(.n.)) ;
```

and from then on simply:

```
tempptr^.tonnage := whatever ;
```

Where different tonnages of different streams have to be accessed, more than one temporary pointer can clearly be allocated.

The standard Pascal function 'WITH' may also be used:

```
WITH str(.n.)^.unmix^.solids^ DO

BEGIN " WITH }

tonnage := whatever ;

" other lines to manipulate the solids record }

END; " WITH }
```

The nesting of WITH statements is not allowed, and the allocation of temporary pointers will, in most cases, be easier to use.

2.6.2.2 Solidvalue.
The data structure to store the different fractions of the tonnage in each of the possible descriptive classes is not simple to use if each fraction is to be accessed individually, rather than all the

fractions in turn. The use of this structure is simplified by the function SOLIDVALUE, which makes the long rows of adjacent arrays take on the appearance of a five dimensional matrix. This further makes the translation of existing FORTRAN subroutines into Pascal much simpler. The function is used in conjunction with the procedure SOLIDSELECT previously described.

If the numerical value of each of the classes in the solid description is known, the fraction may be accessed as follows:

```
tempptr := solidselect(str(.n.)) ;

fraction := solidvalue( 5, 1, 3, 1, 1, tempptr);

{ fraction may now be used, and updated }
```

'FRACTION' will contain the fraction in the fifth size class, the first grade class and the third flotability class. Note that the last two classes need not exist, but then have to be specified as unity.

To replace the updated value of the fraction in the stream record, an equivalent function called SOLIDPUT exists.

2.6.2.3 Mix and split.

Procedures to mix and clear records of streams or the complete streams have also been written. Thus to copy one record into another, the record is first cleared of its original value, and then mixed with the record that it must equal. This makes the writing of some unit operation models extremely simple; for example the solid product substreams of a splitter are simply

copies of the feed stream, *with only the tonnages* corrected. The fractions in each of the classes must obviously remain the same.

2.7 Conclusion.

The data structures required for the description of the general process stream and the accessing of the variables associated therewith can be done in a general and extendable way. The concept of a substream to describe sections of a process stream required to be treated differently from other sections of the stream simplifies the creation of general unit operation models. The data structure designed can accomodate all the requirements of an advanced process simulator stream.

CHAPTER 3 : STREAM TYPE CHARACTERISATION.

3.1 Introduction.

Once the data structures to describe the general hydrometallurgical process stream have been designed, their implementation and use must be considered. The fact that the data structures can adequately describe the process streams and that they can be maintained, does not guarantee their success when simulating a process.

The allocation of the correct stream type, consisting of the correct combination of substreams to accurately describe the process stream, is required for each stream in the process. A stream may be allocated too few, too many or exactly the correct substreams to mimic the true stream.

In terms of simulation, the case where too few substreams are allocated to the simulated process stream, (this including the situation where substreams of incorrect types are allocated and/or not enough substreams that are of the correct type are allocated) the simulation must obviously fail, as the unit operation models will be unable to store the information produced in any memory space. This may be seen as the equivalent of having the pipes in a process plant not matching the unit operations, e.g. a conveyor belt to transport a liquid stream. The underspecification of the substreams of an simulated process stream cannot be tolerated.

The correct allocation of all substreams in all process streams in the flowsheet is, of course, the ideal situation. It is not always possible to predict 'a priori' what the process substream requirements shall be, as these are often unit model, rather than unit operation dependent. As an example, a filter model may or may not require a solid substream description in both its product streams, depending on the accuracy of the model employed. The situation where substream types have to be reallocated every time any changes are made to a flowsheet or any unit operation within the flowsheet is to be avoided if possible.

The overspecification of substreams for any process stream, where the correct substreams are allocated plus some substream(s) not required, is also undesirable. As will be shown, this situation may lead to problems with convergence and will certainly result in the wastage of computer memory. It is however much preferred to the underspecification of substreams, as the simulation may proceed and valid results can be obtained. Unit models will, in general, ignore excess substreams and only operate on those substreams required.

In order to choose a suitable method of stream type allocation, the effects of overspecification of substreams, both on the memory requirements and the convergence properties of the simulator, must be considered.

3.2 Stream Types and Computer Memory.

The overspecification of the substreams required for any process stream must obviously influence the memory requirements of the system. Consider the

flowsheet shown in Figure 3.1, with the "correct" stream types indicated on the *flowsheet*. *Correct* in this case means that for the unit models chosen to *simulate* the process, in each stream only the required substreams have been allocated.

There are two areas of concern in this flowsheet. Should the model for the filters be be substituted with a different model, this new model *allowing the* description of misplaced solids in the clear liquor, a *solid substream* must be allocated for both filter product streams.

The second problem occurs in the solvent extraction section, where an organic phase enters the circuit. No solid nor lixiviant phase substreams are required between the extraction and stripping units. The organic phase substream only *occurs between these two* unit operations. If memory is therefore allocated for *all substreams relevant* to the circuit to all the streams in the circuit, in this case solid, lixiviant and organic substreams, this could result in large memory wastage.

Consider the substreams that are relevant to hydrometallurgy, primarily being solids, carbon, *lixiviant and organic phases*. It can be noted that the solid phase presents the largest number of variables to be described, due to the heterogeneous nature of the population. While comparisons of the number of variables depend on the exact problem under consideration, an order of magnitude difference between the number of variables in solid substreams and lixiviant, carbon and organic *substreams can* be expected. It is thus generally the unnecessary *allocation* of solid substreams that will lead to the greatest storage waste.

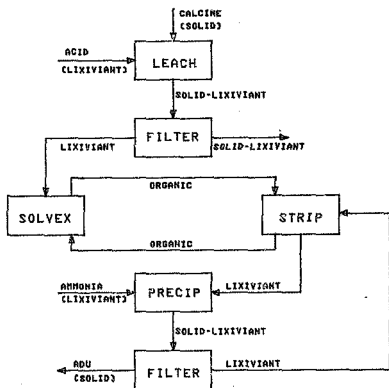


FIGURE 3.1
TYPICAL URANIUM FLOWSHEET

The allocation of a solid substream to every stream in Figure 3.1 will result in a minimum of four excess substreams, if the acid and ammonia system feeds are considered over and above the organic phase. Should the filters not produce solids in the clear liquor output stream, another six streams affected by the filters are unnecessarily assigned solid substreams. Thus from a total of 16 streams, only 6 must contain a solid substream, resulting in a 62% wastage of storage for this particular flowsheet, and considering only the solid phase. Should the filters require solids in all their product streams, the wastage is reduced to 25%.

It is therefore clear that stream type allocation has a large effect on the memory requirements of a simulation. While computer memory is relatively cheap, and the addressable memory of micro-computers is increasing by orders of magnitude, the unnecessary allocation of dynamic memory space for substreams that are not required is not desirable, as the elegance and maintainabilty of the simulator as a whole is reduced.

3.3 Stream Types and Convergence.

A more subtle effect of the substreams allocated to each process stream occurs in the calculation and convergence phase of the simulation. This effect is due to the characteristics of a general simulator; all fields of all correctly allocated substreams of a torn process stream must be initialised before the simulation can continue, and a unit model is only required to update those substreams of its output streams that are affected by the operation of the unit - it cannot be expected of each unit model to

check whether unnecessary substreams have been allocated to *its output streams* and to update them accordingly. This is best illustrated with reference to the simple, hypothetical flowsheet in Figure 3.2.

Assume that all the process streams in Figure 3.2 are taken to be a mixture of of solids and lixiviants and further that the *filter model does not allow any* solids to escape in the clear liquor, i.e. the model assumes that only a lixiviant substream has been allocated to stream number 3. *Let stream 3 be the* torn stream.

The simulation calculations will now proceed as follows:

1. All fields of both the solid and lixiviant substreams of stream 3 are given an initial value for the simulation to proceed, even though only the lixiviant substream is required. (Assume for the moment that these are not all zero's.)

2. Since stream 3 is a tear stream, these initial values are duplicated in tear stream records for comparison, to determine whether convergence has been achieved.

3. The assumed values of stream 3 are used to calculate unit 1. The unit model ignores the excess substream of the feed.

4. The results of unit 1 are used to calculate unit 2. Note that unit 2 only updates the lixiviant substream of stream 3.

5. A convergence check is performed. When the solid substream of stream 3 is compared to the *solid* substream of the tear stream, it appears to

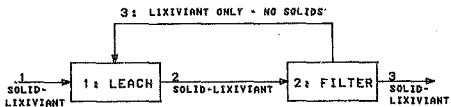


FIGURE 3.2
HYPOTHETICAL PROCESS

have converged, since unit 2 did not change these values.

6. If the lixiviant substream values have not converged, the present value of stream 3 will replace the values in the tear stream, and steps 3 to 6 repeated to convergence.

Once the calculations have been completed, the overall mass balance of the flowsheet will be correct. The mass balances around the individual unit operations will however be incorrect. This is due to the fact that the assumed data for the solid substream allocated to stream 3 is never removed nor updated. Had the substream not been allocated at all the results would of course have been correct.

A few options are available to remove this problem:

1. We may zero all fields of all substreams of tear streams before we recalculate them, at each iteration.
2. As for 1, however this is done only during the first iteration.
3. We may initialise all tear stream to have zero's in all fields of all substreams.

While the third option is by far the simplest to implement, it places an unnecessary restriction on the operation of the simulator. The first and second options are very similar, the second requiring less computer operations as the tear streams are only zeroed once, while the benefits of good initialisation values remain. The second option will however only work when a direct substitution convergence method is used; in more advanced convergence accellerators such as Broyden's method (Broyden, 1965), the effect of historic tear streams

remain to predict the next values to be used, and the torn streams must be zeroed at each iteration step to eliminate this effect. The first option is therefore implemented to ensure that, regardless of the substreams allocated to the tear streams or the convergence accelerator used, the results obtained will not be degraded.

3.4 Stream Type Allocation.

Having noted the effects of incorrect stream type allocation, a method must be implemented to allocate to all streams in the circuit the required substreams to *mimic* the true situation. Three possible methods of stream type allocation are available:

1. User specified stream types.
2. System types.
3. Computer allocated types.

3.4.1.User specified stream types.

The user in this case has to allocate to each stream, whether in the circuit creation phase or by the editing of a data file, a definite stream type consistent with the requirements of the circuit. He must therefore be familiar with the operation of each of the unit models, *their inherent assumptions* and the types of recycle streams. Unit models, in general, are able to accommodate excess substream data, however having an insufficient stream type for calculation will lead to an error condition.

In the ASPEN simulator, each stream in the flowsheet must be allocated a type by the user (Britt, 1984). The simulator further makes use of "class changer" units, to change the type of a stream from one unit to the next when a substream is required to be

removed. Mixers, in ASPEN, are the only unit operation models able to handle different stream types at the same time. Thus the circuit is divided into sections containing only a certain stream type by the user, and the required "type changer" units installed to take care of these changes.

User specification of substreams is the least reliable and most inefficient method of stream type specification. Besides increasing the proficiency required of a user to perform a simulation, it further increases the lag time before results are produced. Users may further rather overspecify data structures, for fear of underspecification and producing an situation impossible to simulate, thus defeating the space saving objective.

3.4.2. System types.

The allocation of a single definitive stream type to the complete circuit being simulated, this type being a combination of all the substreams being used in the flowsheet, leads to the objections expressed earlier, that certain sections of a circuit will be allocated memory for the storage of stream data which does not actually exist in that part of the flowsheet.

The use of system types is however extremely simple, and since the excess memory is allocated dynamically, should not have a detrimental effect on the simulation. The smallest combination of substreams can always be allocated, so that types not used at all in the simulation will not be allocated to any streams.

This method is further fast, and places little or no strain on the user. In cases where the user is dissatisfied with the system *allocation, he may* overwrite this default in the system data file. The correct or overspecification of stream types to all units is guaranteed.

3.4.3. Computer allocated types.

The use of the computer to allocate the stream types is no simple matter, and has two major disadvantages.

Firstly, the allocation of stream types is done as a separate part of the simulation, leading to a time overhead. The determination is iterative and although bound to a maximum of two iterations, leads to requirements of large sections of code to perform these determinations.

In the second instance, this method requires that a "model" be written that *will specify what stream* types will be produced by any unit if presented with a specific feed type. This is time-consuming, and if generality is to be maintained, lengthy. It must further be noted that general unit type allocation *models are not adequate - for* example models of filters may or may not allow for solids to escape in the filtrate, and the specific model type must be used. Any scheme such as this further makes the writing of new models more of a daunting task, a situation that should be avoided.

Note further that under this scheme, a unit model for a specific simulation may not be changed and the calculation repeated without repeating the stream type allocation section of the program as well. The computer allocation of streams allows the checking of

stream types to units before the calculation phase to determine whether any gross unit mismatches have been done. Once again this must be coded in general, difficult for users simply intent on writing a new model.

The main advantage of the computer allocation of stream types is the absolutely optimal usage of computer memory, and the allocation of stream types not visible as feeds to the system. As an example, the organic phase in liquid-liquid extraction, which is recycled internally, is not easily recognizable as a separable phase that must be allowed for only in the internal streams of the flowsheet.

It is primarily the simulation of phase separation, creation and destruction units (e.g. filters, crystallisers and leach units respectively) that makes the characterisation of process streams by automatic means very difficult, if generality is to be maintained.

3.4.4 The Choice of Methods.

Although the preceding sections described three distinctly different methods of stream type allocation, none individually meet the requirements completely. A combination of all three methods is therefore implemented. Whenever possible, the simulator allocates a particular stream type to the circuit as a whole. In certain cases, where the presence of an additional substream may be required for internal recirculation, such as carbon or organic substreams, units may contain additional code to verify that the required substreams have been allocated, warning the user if this has not been done. In all instances, the user must be able to override the substreams allocated by the system.

As an example, in the circuit illustrated in Figure 3.1, allocating a solid-lixiviant stream type to all streams in the circuit will adequately solve the problem, except in the solvent extraction section where an organic phase is recirculated internally. The unit operation calculation procedure should therefore warn the user should the stream type of these streams not include an organic substream. Should users feel that the allocation of a solid substream to streams not containing this substream is too wasteful, they may override this to a stream containing only a lixiviant substream.

In general, as the information required to describe solids is so large, it is the unnecessary allocation of solids storage space that leads to memory wastage. As sections of a flowsheet are often recognisable as being with or without solids, and are not connected by recycles, these sections may be simulated separately, the products of the first simulation being the feeds to the next. In cases where sections are connected by recycles, the presence of solids is generally obvious.

In conclusion, the saving of program size and the gains in the ease of updating models by users themselves point in favour of the allocation of a particular stream type to the complete circuit in question. Should memory allocation be a problem, the user may overwrite the default stream types with the appropriate types. When possible, the system should verify that the stream type allocated will be adequate for the unit operation to be simulated.

CHAPTER 4 : PRECALCULATION ALGORITHMS

4.1 Introduction

In order to coordinate the simulation computations for each of the process units, the specification of a precedence-order in which each subroutine must be computed is required. This involves breaking the complete flowsheet into groups of units or sub-plants which interact such that only a forward flow of information occurs. This is referred to as partitioning the system. These partitioned groups are also referred to as maximal cyclical nets, or if they consist of groups of unit operations, as unit maximal cyclical loops. (UMCL's). Ford (1976) showed that partitioning a system so that each UMCL is solved sequentially in the order of mass flow does not degrade the convergence behaviour of the system even though the problem size is considerably reduced. Considering each UMCL in turn, the order of calculation of each unit operation model within the UMCL must be determined.

A feed stream to any unit process which is not known the first time a unit operation is computed, is known as a recycle stream. If recycles are present in a UMCL, tearing is required. This involves the opening or "tearing" of streams such that all cycles in the process are removed. Initial values for the variables present in tear streams are assumed and an iterative scheme is employed to force the convergence of the torn stream to within a specified tolerance.

Once all of the tear streams have been determined, a calculation order for all units in the UMCL may be determined. This calculation path is then repeated sequentially until convergence of all the tear variables has been achieved. It is this sequential calculation of all the unit operations to convergence that is implied by the term "sequential modular simulator".

4.2 Partitioning the flowsheet.

Partitioning the flowsheet into cyclical nets or UMCL's that may be solved individually may be done by performing a path-search. This is presently considered to be the most effective way of partitioning (Evans, 1980). Path-searching requires a sequential search of connections between units to be done until no further common paths (or recycles) are found. This group of units containing the common connecting paths is known as a cyclical net or UMCL. Path searching algorithms operate on a graphical representation of the flowsheet in which the units are considered as nodes and the streams as vertices of a directed graph.

The path-searching algorithm of Sargent and Westerberg (1964) in the modified form as independently rediscovered and presented by Tarjan (1972) is very efficient and robust. This algorithm is used for partitioning in the ASPEN chemical process simulator and is considered to be the best currently available (Evans, 1980). The algorithm as presented by Tarjan further operates recursively, using a depth-first search of the directed graph. A depth-first search uses the following rule: When selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which

still has unexplored edges (Tarjan, 1972). This recursive technique makes the Pascal implementation particularly simple and elegant. The Tarjan algorithm was implemented for use in the simulator.

The Tarjan algorithm operates in O(v,e) time, where v is the number of vertices and e the number of edges in the graph. Two stacks are maintained during the search, one to store the current component under consideration, the other to keep track of points that have been reached during a search but have not become part of a current component. A current component includes a vertex if the vertex is found to have been considered before and is connected to the current component by another edge.

Once the units in each UMCL have been determined, the stream included in that net are found by considering the product streams of each unit in turn. Those streams which are feeds to other units in the UMCL are clearly included in the net and considered for possible tearing as they form part of a cycle. Product streams not feeding units included within the present UMCL are not further considered.

For the determination of the calculation order of the UMCL's, the feed and product streams of the net as a whole are required. These are determined as an extension of the procedure to determine the streams included in each UMCL; those unit product streams not forming part of this UMCL must clearly be products of this UMCL, these product streams are further feeds to the UMCL's that contain the units fed by these streams. System feeds and products are left out of consideration in all cases.

Once all the feeds and products of all UMCL's in the system are known, the UMCL calculation order may be determined. As system feeds and products are not considered, the first UMCL calculable must have "zero" feeds, i.e. no feeds from other UMCL's. The product streams from this UMCL may now be considered known, and deleted from the feed stream lists of the remaining UMCL's. As each successive UMCL is found, the tear streams within that UMCL are determined and the calculation order of each unit within the UMCL calculated. This procedure is repeated until all the UMCL's have been ordered.

Although the determination of the streams contained within any UMCL is based on the extensive searching of arrays of unit feed and product stream lists, the procedure is fast and reliable.

4.3 Tearing the flowsheet.

In the iterative procedure to solve a UMCL consisting of more than one unit operation, each unit must be calculated once per iteration. All the inputs to a unit must be known before the outputs can be determined. If the UMCL contains recycles, as it must if it consists of more than a single unit operation, it is necessary to assume some input streams in order to start the calculation procedure. These assumed streams are termed "tear streams".

A valid decomposition is a set of tear streams that opens all cycles in the UMCL at least once. A redundant decomposition is a valid decomposition from which at least one stream may be removed without rendering the resulting decomposition invalid; a nonredundant decomposition has no such stream.

Once the process computation is started, the particular unit ordering (or one of its cyclical permutations) repeats itself during successive iterations. In any complex system of unit operations there are possible permutations of the order in which the unit operations can be calculated. These permutations depend on the choice of the tear streams. As the order in which the unit operations are calculated ultimately influence the convergence characteristics of the system as a whole (Upadhye and Grens, 1975), tear streams must be chosen that will allow convergence in the least possible computer time, using the least storage of information in memory.

There is no difference, as far as convergence of the process computations are concerned, between the cyclic permutations of any unit ordering. Different valid decompositions may therefore result in the same convergence characteristics. Such different tear sets leading to the same convergence are known as a family of tear sets.

The question of an optimal tear set was investigated by Upadhye and Grens (1975), and it was shown that the convergence rate of any flowsheet will be reduced if a valid decomposition from a family of tear streams other than the nonredundant family is chosen. They further showed that in most cases, to choose the best decomposition for direct substitution convergence, it is only necessary to find the nonredundant decomposition family and to select any convenient decomposition in that family.

Many algorithms exist to find the nonredundant family of valid tear sets, these being based on either exact or probability methods. While the latter method has

been greatly favoured, it is not guaranteed to find a solution, and in certain cases a *"branch and bound"* section is required based on a bound trial and error search for the optimal solution. An algorithm implementing this procedure (Motard and Westerberg, 1981) was used in the ASPEN simulator (Evans, 1980). Other probability algorithms include those of Pho and Lapidus (1973) and Christensen and Rudd (1969). Ford shows the former to be ineffective.

Of the exact methods, the algorithm of Upadhye and Grens (1972) based on dynamic programming has been used by MODSIM (Ford, 1976). This algorithm is particularly effective, and is guaranteed to find the optimal solution in all cases. In this algorithm, streams may further be weighted to allow the selection of different tear sets from the nonredundant family based on various criteria, such as the *minimum number of stream* variables that need to be assumed (Ford, 1976).

The method of Upadhye and Grens does however suffer major disadvantages. The algorithm sequentially investigates a set of possible states, each state representing the opening of a combination of all cycles in the circuit. As the *number of possible* states is 2^n-1, where n is the number of cycles in the circuit, and each state requires at least three real storage positions (or 18 bytes of memory), this results in huge space requirements. A more subtle restriction is however the largest number accurately representable by any computer. Each state requires an integer reference number to indicate its position in an array of states. As $2^{16}-1$ is generally the largest integer accurately representable by most small computers, the algorithm is therefore restricted to systems consisting of 16 or fewer cycles. Note that

using negative integers to double the number of states representable only increases the maximum number of cycles by one, with a corresponding doubling of the required storage space.

In ore-dressing simulation, circuits containing more than six cycles in any complex node (UMCL) are rare, while in hydrometallurgical simulation a five unit countercurrent circuit may have as many as forty cycles, due to the multiple feed-product nature of the unit operations. The exact method of Upadhye and Grens is therefore not suitable for a general simulator.

One other exact method which has often been rejected from consideration due to its large combinatorial problem is that of Lee and Rudd (1966). This algorithm is based on the concept of "containment" of streams and sets of streams in other streams and sets of streams. If one stream opens the same or some of the cycles that any other stream opens, it is included in the latter, and may be eliminated from further consideration. The same concept may be applied to sets of streams and their inclusion in other sets of streams. For systems with with large numbers of streams, a large number of sets must be tested for containment, resulting in severe combinatorial problems. The use of FORTRAN for the manipulation and creation of large numbers of sets further complicates coding, and may have lent further impetus to the development of alternative tearing algorithms.

Pascal allows the definition of the set as a data type. The set operations for addition, difference and intersection further exist. The manipulation of sets may therefore be done efficiently. The Lee and Rudd

algorithm was hence further investigated and subsequently modified to reduce the combinatorial problem described. This modified algorithm finds the smallest valid tear set that opens the minimum number of cycles. In all cases tested, this modified algorithm produced a tear set from the nonredundant family of tear streams. The algorithm as implemented has no bounds on the number of cycles in any UMCL that may be operated on.

For both the Upadhye and Grens and the Lee and Rudd algorithms, as well as its modified version, the stream-cycle matrix is required. This is essentially the list or set of cycles that each stream in the circuit is included in. The determination of the stream-cycle matrix is discussed in section 4.4.

The modified Lee and Rudd algorithm as implemented for the present simulator operates as follows:

1. Investigate each stream in the stream-cycle matrix for containment; this being defined as one stream being included in a set of cycles smaller or equal to the set of cycles that contains any other stream. If contained, the stream is removed from further consideration. This stage reduces the problem size.

2. If any cycle now contains only a single stream, it is a self-loop, and the stream must be a tear stream. If such a cycle exists, eliminate that cycle as well as all other cycles that contain this tear stream from further consideration. Return to step 1.
If no streams or cycles may be further eliminated, yet all cycles have not been removed, proceed.

3. *For each remaining* cycle, a nonvalid tear set exists, being that set of streams not containing any of the streams in the cycle. Clearly tearing only this set will not open all the cycles in the circuit. Each possible combination of streams remaining can now be tested for inclusion in the nonvalid tear set of each remaining cycle. If included, it is itself a nonvalid set, *if not included in any nonvalid* tear set, it is a valid tear set.

4. Combinations of the streams remain ; as a possible tear set are tested in increasing set size. Once a valid set is found, all combinations of remaining streams with the same set size are tested to find that smallest tear set opening the fewest number of cycles. This satisfies the Upadhye and Grens criterion for determining a tear set from the nonredundant family of tear sets.
This set of streams, combined with those tear streams found during step 2, constitute the complete tear set. In most simple cases, the complete tear set will be found without having to proceed with steps 3 and 4.

The validity of this modified algorithm rests on one assumption; that the redundant families of tear streams shall not contain a valid tear set with fewer tear streams than the smallest valid tear set from the nonredundant family. This assumption is required as the algorithm first determines the minimum size of a valid tear set, then proceeds to find that valid tear set opening the *least number of cycles*.

The requirement of this assumption is further evident when the algorithm is compared with that of Upadhye and Grens (1972), in which any tear set from the nonredundant family could be produced, the valid set

with the smallest number of torn streams being found by the application of their "Replacement Rule" which states that if all the input streams to any unit in the process are included in a tear set, they may be replaced with the output streams of that unit, without affecting the validity of the decomposition or the direct substitution convergence properties.

The assumption mentioned is only required for the algorithm as implemented. It is of course possible to search all possible sets of streams that can constitute a valid tear set for the optimal solution, this being very time-consuming. An upper-bound on the size of the largest set of tear streams exists, this being the number of cycles in the circuit divided by the least number of cycles opened by any stream in the circuit. Should any doubt exist about the validity of the tear set produced by the algorithm as implemented, this fail-safe search method may be used, searching all possible tear sets having a size smaller or equal to the possible maximum size.

A comment on the weighting of streams during the tearing phase is required. In the Upadhye and Grens (1972) algorithm, streams may be weighted such that the nonredundant tear set found shall also contain the minimum number of stream variables (Ford, 1976), in an attempt to reduce the number of variables that need to converge during the calculation phase. This implies an a priori knowledge of the type of stream under consideration, e.g. whether a stream contains solids or not. As discussed in section 3.3, three methods for the allocation of stream types exist; an overall type for the complete circuit, user specified stream types for each individual stream and the determination of the required stream types by an initial simulation using pseudo-unit operations that

determine the output stream type produced by any unit operation given the feed stream types.

For the first of these methods, as all streams shall have an equal number of variables, the set of tear streams produced will not be affected by the assigned weights, while using the last method requires a set of tear streams for the pseudo-*simulation*, *the stream* types obtained being used to determine the second tear set, this being used for the true simulation calculations. This method will not only increase the program size considerably, but will reduce the maintainability of the program. The unsuitability of this method for the allocation of stream types was discussed earlier (Section 3.3), the requirement of two different tearing algorithms further proves the point. Only in the case where the user explicitly defines the type of each stream in the circuit may convergence be enhanced by the weighting of streams during the tearing procedure. As the time required to define the stream types may be substantial (and error prone), the resultant reduction in calculation time may very well be lost.

The weighting of streams during the tearing phase should therefore be restricted to those weights required to produce the minimum number of tear streams from the nonredundant family. This only requires knowledge of the number of cycles opened by each stream in the circuit - a result known from the stream-cycle matrix determined (Section 4.4).

4.4 Determination of the stream-cycle matrix.

In order to use either of the exact tearing algorithms discussed previously, the stream-cycle matrix is required. This is essentially a list of the

cycles in the flowsheet or UMCL, and the streams that make up each cycle.

Mateti and Deo (1976) review all types of algorithms for enumerating all the cycles of a directed graph or flowsheet. From their comparative work, only those algorithms based on a backtracking and path-searching approach are polynomially time bound, while methods using circuit vector space, the edge digraph and powers of the adjacency matrix are exponential in their enumeration time. An algorithm using backtracking and path-searching was therefore implemented.

Backtrack algorithms search for cycles in the super set of all cycles and their cyclic permutations in the flowsheet. The efficiency of the algorithms depend on

1. The size of the super set,
2. The effort required to compute an element in the super set of cycles, and
3. A test to determine whether an element is indeed a cycle. (Mateti and Deo, 1976)

The enumeration algorithm of Tarjan (1973) uses improved pruning techniques to decrease the size of the subset of possible cyclic paths considerably. This algorithm operates recursively in a time bound of O(n.e.c), where n is the number of vertices (or units), e the number of edges (or streams) and c the number of cycles in the UMCL under investigation. This algorithm was implemented for the enumeration of the cycles in the flowsheet, primarily for its elegance, robustness and simplicity.

Tarjan describes the algorithm as follows: "The point stack used in the algorithm denotes the elementary path p currently being considered; the elememtary path has has start vertex s. Every vertex v on such a path must satisfy v >= s. A vertex v becomes marked when it lies on the current elementary path p. As long as v lies on the current elementary path or it is known that every path leading from v to s intersects p at a point other than s, v stays marked.

For each vertex s, the algorithm generates elementary paths which start at s and contain no vertex smaller tjan s. Once, a vertex v has been used on a path, it can only be used to extend a new path when it has been deleted from the point stack and when it becomes unmarked. A vertex v becomes unmarked when it may lie on a simple circuit which is an extension of the current elementary path. Whenever the last vertex on an elememtary path is adjacent to the start vertex s, the elementary path corresponds to an elementary circuit which is enumerated."

Other algorithms based on backtracking that are theoretically faster than the Tarjan algorithm, being time bound to O((n+e)c), include those of Johnson (1975) and Tarjan and Reid (1976). These algorithms were not used as the increase in speed produced was not considered adequate to justify the increased programming effort. It has been found during verification of the the simulator (Chapter 5), that the relative time taken to perform all the precalculation algorithms is negligible compared to the time required to calculate to convergence any simple circuit that requires tearing; simplicity and maintainability of the coded algorithms being of greater importance than speed.

4.5 Determination of the calculation order of units within a *UMCL*.

Once all the tear streams in a UMCL are known, the calculation cycle that is required to be repeated to convergence must be determined. This cycle is essentially the sequential calling of each calculation subroutine that models any particular unit operation. The determination of the calculation cycle is search based, and operates as follows:

Consider each unit operation in the UMCL in succession. A unit of which all the feeds are tear streams or system feeds must exist (if we have determined a valid tear set). This unit is thus placed in the calculation order cycle, and its product streams are removed from the feed lists of the remaining units. The search for a unit having *only known streams as* feeds is repeated until all the units have been placed in the calculation sequence. Repeating the sequence thus determined will lead to the convergence of all tear streams in the UMCL.

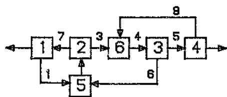
4.6 Verification.

In order to verify the operation of the *tearing and* ordering algorithms, a large number of tests were performed, both on industrial and hypothetical circuits. Of those tests performed, the examples used by Upadhye and Grens (1975) are shown to allow comparison with future work. Further to these a hypothetical hydrometallurgical countercurrent liquid-liquid extraction example was simulated, which indicates the limitations of the dynamic programming approach.

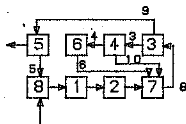
The example flowsheets are shown in Figures 4.1 a-d, while the results obtained by the modified Lee and Rudd algorithm and Upadhye and Grens (1975) are summarised in Table 4.1. From these results it can be noted that in all cases the smallest nonredundant tear set was produced by the new algorithm, and that the problem size that can be accomodated by this algorithm exceeds that of dynamic programming. In all cases a viable calculation order was produced, as is shown.

4.7 Conclusion.

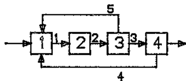
The algorithms implemented have been tested using true simulation examples. In all cases the algorithms have produced the optimal tear set and the correct unit calculation order. The algorithms are able to accomodate any number of units, process streams and cycles within any recycle net. The algorithms are therefore adequate for the simulation of hydrometallurgical circuits, and in general, any process circuit.



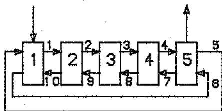
a. CAVETT'S PROCESS



b. ROSEN'S PROCESS



c. RAMJI'S PROCESS



d. STRIP PROCESS

FIGURE 4.1
EXAMPLE FLOWSHEETS
FOR TEARING AND ORDERING

PROCESS	NONREDUNDANT TEAR SETS - UPADYE AND GRENS (1975)	TEAR-SET MODIFIED LEE AND RUDD ALGORITHM	CALCULATION ORDER
CAVET	{2,5} {2,8} {5,6,7}	{4,1} (Equivalent by replacement rule)	3,4,5,2,6,1
ROSEN	{8} {5,6,10}	{8}	3,5,4,6,8,1,2,7
RAJI	{1} {2} {3,5}	{1}	2,3,4,1
STRIP	Problem too large	{1,2,3,4,6}	5,4,3,2,1,

Table 4.1

Verification of modified Lee and Rudd Algorithm

CHAPTER 5 : *CONCLUSIONS*

5.1 Introduction

The future of simulation in the minerals industry is assured. Simulation for design and optimisation has become an active field. Modelling of *hydrometallurgical processes using* the population balance approach is being researched and new models for unit operations are constantly being developed. It is essential that this research being done in *different* fields of metallurgy be united, so that the simulation of complete metallurgical circuits may be done with the best available models. This can only be accomplished if a maintainable and extendable simulation executive exists that can combine the models into a complete process. It was the aim of this research work to produce such a simulation executive.

5.2 Summary

5.2.1 Data structures

Data structures were designed to represent the general process stream in a maintainable and extendable way, so that it may be used for the simulation of any hydrometallurgical operation. The data is easily accessible to the user, and the creation of new process operation models is a simple task. The accessing and replacing of variables operated on by a unit operation model is simplified by the use of utility routines that make these data structures appear totally transparent.

The use of the concept of substreams allows simple additions to the stream data structures, so allowing the future extension of the simulator to other branches of metallurgy, such as pyrometallurgy. Existing substreams can be extended by additions to the descriptive record.

The use of an advanced structural language such as Pascal has resulted in the dynamic allocation of memory, thus utilising the computer storage more effectively in all sections of the program. Dynamic memory allocation in stream descriptive records further allows a practically unlimited number of variables to be represented.

The overall stream data structure is considered adequate and extendable enough to accomodate all hydrometallurgical simulations.

5.2.2 Precalculation algorithms.

The precalculation algorithms utilise procedures that are general enough to operate on any flowsheet. No limit exists on the number of units or streams in the process, and the simulator will produce a tear set that shall allow direct substitution to converge in the least number of iterations. Tests have indicated that the time taken to tear and order the flowsheet is negligable compared to the calculation phase of the flowsheet.

The precalculation algorithms are considered adequate for any further extensions that may be done to the simulator executive.

5.2.3 Verification

A large number of ore-dressing unit operation models were taken from the MODSIM simulator and translated into Pascal to be used in the present executive. The unit models translated were:

1. Separation Units

    a. Classification Units

        i. Cyclone
        ii. Screens

    b. Coal Washing Units

        i. Drewboy washer
        ii. Dense Medium Hydrocyclone

    c. Bank of Flotation Cells

    d. Gravity Concentration Units

        i. Spiral
        ii. Reichert Cone

2. Comminution Units

    a. Grinding Mills
    b. Crushers

3. Water Separation Units

    a. Thickeners

Comparitive test were performed between the simulators, the present version yielding the same results as MODSIM in all cases. The example flowsheets tested are shown in Figures 5.1 a and b, the results obtained for the two simulations are summarised in Table 5.1. This verification was required to validitate the usability of the data structures and the effectiveness of the precalculation algorithms.

Extensions to the stream structures and unit operation models was done by Stange (1985) for carbon in pulp research. Simulation of this hydrometallurgical operation was done successfully, illustrating the extendability of the simulator to the exact requirements of the ultimate user.

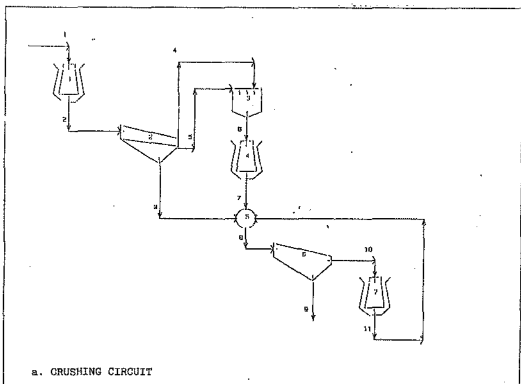
5.3 Future research

5.3.1 Introduction

The simulator in its present form can successfully simulate a large number of metallurgical processes. Ore-dressing models for the simulator have been written and tested, as well as a carbon in pulp system. For the simulation of all the *hydrometallurgiacl* unit operations, future work is required.

5.3.2 *Unit operation* models.

Research is in progress on the creation of *hydrometallurgical unit operations*. This field has received new impetus from the work of Sepulveda (1978) with the population balance approach to *leaching*. *This approach* is being extended to other hydrometallurgical operations.



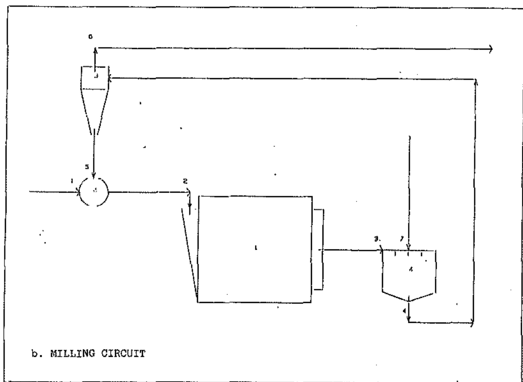


FIGURE 5.1 FLOWSHEETS FOR SYSTEM VERIFICATION

Strm No	Mass Flowrate TPH
1	2500.0000
2	2500.0000
3	566.8865
4	1203.0377
5	730.0758
6	1933.1135
7	1933.1135
8	4452.7446
9	2501.0843
10	1952.7446
11	1952.7446

SIZE (microns)	% Passing Stream 1	% Passing Stream 9
1220000	99.93	100.00
862678	96.94	100.00
610011	81.10	100.00
431347	54.91	100.00
305011	31.66	100.00
215677	16.64	100.00
152508	8.33	100.00
107841	4.07	100.00
76255	1.97	100.00
53921	0.95	100.00
38128	0.45	100.00
26961	0.22	100.00
19064	0.10	100.00
13480	0.05	100.00
9532	0.02	75.49
6740	0.01	53.11
4766	0.01	40.05
3370	0.00	31.56
2383	0.00	21.96

TABLE 5.1 RESULTS OF SIMULATIONS
a. CRUSHING CIRCUIT

Strm No	Mass Flowrate TPH	Water Flow TPH	Percentage Solids
1	100.0000	233.3333	30.00
2	453.0672	381.6099	54.28
3	452.1535	381.6278	54.23
4	452.1535	712.6304	38.82
5	352.6700	148.2766	70.40
6	99.4835	564.3539	14.99
7	-	331.0026	-

SIZE (microns)	% Passing Stream 1	% Passing Stream 5
2360.0	97.60	99.73
1700.0	94.50	99.03
1180.0	90.50	97.79
850.0	85.50	95.67
600.0	78.90	92.17
425.0	69.80	86.24
300.0	56.70	76.33
212.0	40.30	60.40
150.0	27.60	40.38
106.0	20.00	22.79
75.0	15.10	11.94
53.0	11.40	6.48
38.0	8.60	3.88
27.0	6.80	2.59

TABLE 5.1 RESULTS OF SIMULATIONS

b. MILLING CIRCUIT

The simulation of carbon-in-pulp and carbon-in-leach processes is of great importance at present, and it can be expected that research in these fields will be active in the next few years.

The testing and verification of models developed in research is made simple by their inclusion in an existing simulator. In this way their interaction with other unit operations and in recycle situations may further be tested.

5.3.2 *Convergence*

It has been shown by Westerberg et al (1979) that direct substitution will always lead to the convergence of a realistic process flowsheet. The rate of convergence may however be slow, and may be prohibitively so on a microcomputer.

The use of convergence accelerators is a critical part of the research required to be performed in future. A distinct problem with the Newtonian accelerators available at present is that the derivatives of the product process streams of each unit operation is required, which is a function of the unit operation model. These derivatives may therefore be extremely complicated and their calculation could be as time-consuming as the direct-substitution process itself; the decrease in the number of iterations being lost in the increased calculation effort.

The bounded Wegstein convergence accelerator has further been implemented, (Westerberg et al, 1979) this method generally yielding improved convergence characteristics.

The robustness of direct substitution makes it particularly suitable for development work, and was therefore retained. It was further felt that the convergence problem may only be optimally investigated once more knowledge is available on the nature of the unit operation models, and should be postponed until that time when it may form the centre of more intensive research in that field.

5.3.3 Human-computer interfaces

The larger interfaces that will link the user and the simulator have been written but were not part of the present investigation. These will create flowsheets and produce graphical and numerical output in a useable form. The maintainability and modularity of the sections already created makes the writing of these interfaces quite a simple matter, the interaction of the user and the machine becoming the major design criterion. The final version of the interface should allow simulations to be done by any user with no computer skills.

5.4 Conclusion

The simulator executive created during this research project fulfills all the requirements of an advanced hydrometallurgical simulator.

The simulator is considered general, maintainable and extendable enough to accomodate any hydrometallurgical process operation model.

REFERENCES

Britt,H.I. (1980) Multiphase stream structures in the ASPEN process simulator. In Foundations of *computer-aided chemical process design*, Vol 1, 471-510, R.S.H.Mah and W.D.Seider (eds), Engineering Foundation, New York.

Broyden,C.G. (1965) A class of methods for solving nonlinear simultaneous equations. Math. Comput. 19, 577.

Christensen,J.H. and D.F.Rudd (1969) Structuring design computations. AIChE J. 15, 94-100.

Evans,L.B. (1980) Advances in process flowsheeting systems. In Foundations of computer-aided chemical process design, Vol 1, 425-468, R.S.H.Mah and W.D.Seider (eds), Engineering Foundation, New York.

Evans,L.B. and W.D.Seider (1976) The requirements of an advanced computing system. CEP June, 80-83.

Evans,L.B., B.Joseph and W.D.Seider (1977) System structures for process simulation. AIChE J. 23, 658-666.

Evans,L.B. et al (1979) ASPEN: an advanced system for process engineering. Comput.Chem.Eng. 3, 319-327.

Ford,M.A. (1976) Simulation of ore dressing plants. PhD thesis, University of the Witwatersrand, Johannesburg.

Ford,M.A. and King,R.P (1984) The simulation of ore-dressing plants. Int.J.Min.Proc. 12, 285-304.

Hess,F.W. and Wiseman,D.M. (1984) Interactive colour graphics process analyser and simulator for mineral concentrators. 18th International APCOM Symposium, London, England, 26-30 March.

Johnson,D.B. (1975) Finding all the elementary circuits of a directed graph. SIAM J.Comput. 4, 77-84.

Kaijaluoto,S. (1979) Experiences in the use of plex data structure in flowsheeting simulation. Comput.Chem.Eng. 3, 307.

Lee,W. and Rudd,D.F. (1966) On the ordering of recycle calculations. AIChE J. 12, 1184-1190.

Mateti,P. and Deo,N. (1976) On algorithms for enumerating all circuits of a graph. SIAM J.Comput. 5, 90-99.

Motard,R.L and A.W.Westerberg (1981) Exclusive tear sets for flowsheets. AIChE J. 5, 725-732.

Neville,J.M. and Seider,W.D. (1980) Coal pretreatment - extensions of FLOWTRAN to model solids handling equipment. Comput.Chem.Eng. 4, 49-61.

Pho,T.K. and L.Lapidus (1973) Topics in computer aided design : I. An optimum tearing algorithm for recycle streams. AIChE J. 19, 1170-1181.

Richardson,J.M., D.R.Coles, J.L.Vanderbeek and J.W.White (1980) Flexmet - a flexible computation system for steady-state analysis of metallurgical

process flowsheets. AIME Arizona conference meeting, December.

Richardson,J.M., D.R.Coles and J.W.White. (1981) Fluor Mining and Metals introduces *Flexmet*, a computer-aided and flexible metallurgical technique for steady-state flowsheet analysis. Eng.Min.J. 182, 88-97.

Ritchie,I.C. (1984) Economic evaluation of minerals extraction processes by use of a flexible process simulation program. PhD thesis, Royal School of Mines London.

Ritchie,I.C. and Spencer,R. (1984) Economic evaluation of minerals extraction processes by use of a flexible process simulation program. 18th International APCOM symposium, London, England, 26-30 March.

Sargent,R.W.H. and A.W.Westerberg (1964) "Speed-up" in chemical engineering design. Trans. Instn. Chem. Engrs., 42, 190-197.

Sepulveda,J.E and Herbst,J.A (1978) A population balance approach to the modeling of multistage continuous leaching systems. AIChE Symposium series, *Fundamentals of* hydrometallurgical operations.

Stange,W. (1985) MSc. thesis in preparation. University of the Witwatersrand, Johannesburg.

Tarjan,R. (1972) Depth-first search and linear graph algorithms. SIAM J. Comput. 1, 146-160.

Tarjan,R. (1973) Enumeration of the elementary circuits of a directed graph. SIAM J.Comput., 2, 211-216.

Tarjan,R. and Reid,R.C (1976) Bounds on backtrack algorithms for listing cycles, paths and spanning trees. Networks, 5, 237-252.

Upadhye,R.S. and E.A.Grens II (1972) An efficient algorithm for optimal decomposition of recycle streams. AIChE J. 18, 533-539.

Upadhye,R.S. and E.A.Grens II (1975) Selection of decompositions for chemical process simulation. AIChE J. 21, 136-143.

Westerberg,A.W. et al (1979) Process flowsheeting. Cambridge University Press, Cambridge.

Author Cilliers Johannes Jacobus Le Roux
Name of thesis Hydrometallurgical Simulation - A Viable Program Structure. 1986

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2013

LEGAL NOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.