



# Object-Oriented Simulation of Chemical and Biochemical Processes

Damien Hocking

Department of Chemical Engineering  
University of Adelaide

Thesis submitted for the Degree of  
Doctor of Philosophy  
in  
The University of Adelaide  
Faculty of Engineering

February 1997

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signed:

Damien Hocking

Date: 2<sup>nd</sup> July, 1997

# ACKNOWLEDGMENTS

I would like to thank my supervisors, Dr. Brian O'Neill and Dr. John Roach for their support and guidance throughout this project. Their contribution has been invaluable. I would also like to thank the staff and postgraduate students of the Chemical Engineering Department at the University of Adelaide for many interesting discussions and assistance. I would also like to thank my family and friends for their support.

I gratefully acknowledge the financial support of the Co-operative Research Centre for Tissue Growth and Repair without which I could not have commenced this project.

# CONTENTS

|  |           |
|--|-----------|
| <b>Chapter 1: Introduction and literature review</b>       | <b>1</b>  |
| 1.1 Simulation Techniques                                  | 1         |
| 1.1.1 Sequential-Modular                                   | 1         |
| 1.1.2 Equation-Oriented                                    | 2         |
| 1.1.3 Parallel-Modular                                     | 4         |
| 1.2 Object-Oriented Process Simulation                     | 4         |
| 1.2.1 Object-Oriented Simulation                           | 6         |
| 1.2.2 Languages  | 6         |
| 1.2.3 Object-Oriented Simulation Environments              | 14        |
| 1.2.4 Summary of Object-Oriented Simulation                | 18        |
| 1.3 Biochemical Process Simulation                         | 19        |
| 1.3.1 Summary of Biochemical Process Simulation            | 21        |
| 1.4 Physical Property Calculation                          | 22        |
| 1.5 Numerical Analysis Methods                             | 24        |
| 1.5.1 Nonlinear Algebraic Equations                        | 24        |
| 1.5.2 Integration Methods                                  | 27        |
| 1.6 Conclusions and Project Scope                          | 29        |
| <br>   |           |
| <b>Chapter 2: Simulator Development and Data Structure</b> | <b>31</b> |
| 2.1 Development Language                                   | 31        |
| 2.2 Data Structure   | 35        |
| 2.2.1 Physical Information                                 | 36        |
| 2.2.2 Simulator Executive                                  | 39        |
| 2.2.3 Mathematical Information                             | 40        |
| 2.3 Functionality and Behaviour                            | 48        |
| 2.3.1 Structural Analysis                                  | 48        |
| 2.3.2 Equation Evaluation                                  | 51        |
| 2.3.3 Model Evaluation                                     | 52        |
| 2.3.4 Behavioural Changes                                  | 54        |
| 2.3.5 Numerical Methods                                    | 56        |
| 2.3.6 Interchangeable Simulation Techniques                | 60        |

|  |     |
|--|-----|
| 2.4 Chemical Components and Property Calculation                   | 62  |
| 2.5 Summary  | 66  |
| <b>Chapter 3: C++ Implementation</b>                               | 68  |
| 3.1 C++ Constructors and Destructors                               | 68  |
| 3.2 Vectors and Matrices   | 69  |
| 3.3 Process Class Structure  | 73  |
| 3.3.1 System Class and Descendants                                 | 73  |
| 3.3.2 Port Class and Descendants                                   | 77  |
| 3.3.3 Stream Class and Descendants                                 | 80  |
| 3.4 Mathematical Class Structure                                   | 81  |
| 3.4.1 Variable Class and Descendants                               | 81  |
| 3.4.2 Equation_Set and Dynamic_Set classes                         | 83  |
| 3.5 Component, General_Component_Mixture<br>and Properties Classes | 88  |
| 3.5.1 Component class and Descendants                              | 89  |
| 3.5.2 General_Component_Mixture Class                              | 90  |
| 3.5.3 Properties Class and Descendants                             | 90  |
| 3.6 Numerical Method Classes                                       | 91  |
| 3.7 Summary  | 92  |
| <b>Chapter 4: Modelling and Simulation</b>                         | 93  |
| 4.1 Decomposition Techniques                                       | 93  |
| 4.1.1 Medium and Machine Decomposition                             | 93  |
| 4.1.2 Primitive Behaviour Decomposition                            | 94  |
| 4.1.3 Mathematical Decomposition                                   | 95  |
| 4.2 Modelling Examples   | 97  |
| 4.2.1 Mixing Tank  | 98  |
| 4.2.2 Bi-Directional Information Flow                              | 105 |
| 4.2.3 Connected-System Modelling                                   | 112 |
| 4.2.4 Multiple-Inheritance Modelling                               | 120 |
| 4.2.5 Modelling with Physical Properties                           | 126 |
| 4.3 Simulation   | 130 |

|                     |   |     |
|---------------------|---|-----|
| 4.3.1               | Instruction Sequence                            | 130 |
| 4.3.2               | Steady-state example                            | 131 |
| 4.4                 | Summary   | 135 |
| <b>Chapter 5:</b>   | <b>Major Test Problems</b>                      | 136 |
| 5.1                 | Cavett Problem                                  | 136 |
| 5.2                 | Tennessee Eastman Process                       | 141 |
| 5.2.1               | Control Systems                                 | 143 |
| 5.2.2               | Simulation Results                              | 150 |
| 5.3                 | Recombinant Fermentation Model                  | 157 |
| 5.3.1               | Model Description                               | 158 |
| 5.3.2               | Control System                                  | 161 |
| 5.3.3               | Simulation Results                              | 162 |
| 5.4                 | Discussion                                      | 165 |
| 5.5                 | Summary   | 167 |
| <b>Chapter 6:</b>   | <b>Summary, Conclusions and Recommendations</b> | 168 |
| 6.1                 | Summary   | 168 |
| 6.2                 | Class Description                               | 168 |
| 6.3                 | Modelling                                       | 169 |
| 6.4                 | Simulation                                      | 170 |
| 6.5                 | Recommendations                                 | 171 |
| <b>Bibliography</b> |   | 173 |
| <b>Nomenclature</b> |   | 180 |
| <b>Appendices</b>   |   | 182 |
| <b>Appendix A:</b>  | <b>General member function descriptions</b>     | 183 |
| A.1                 | <b>System-based classes</b>                     | 183 |

|   |         |
|---|---------|
| A.1.1 <b>System</b> Connectivity and<br>Mathematical interface functions                    | 183     |
| A.1.2 <b>System</b> Analysis  | 185     |
| A.1.3 <b>Convergence_Block</b> class interfaces   | 185     |
| A.2 <b>Port</b> -based classes  | 186     |
| A.2.1 <b>Port</b> , <b>Input_Port</b> and <b>Output_Port</b><br>class interface functions   | 186     |
| A.2.2 <b>Process_Output_Port</b> and<br><b>Process_Input_Port</b> class interface functions | 187     |
| A.2.3 <b>Signal_Input_Port</b> and<br><b>Signal_Output_Port</b> class interface functions   | 189     |
| A.2.4 <b>Energy_Input_Port</b> and<br><b>Energy_Output_Port</b> class interface functions   | 190     |
| A.3 <b>Stream</b> classes   | 190     |
| A.3.1 <b>Stream</b> class interface functions   | 190     |
| A.4 <b>Variable</b> -based classes  | 190     |
| A.4.1 <b>Variable</b> class interface functions   | 190     |
| A.4.2 <b>Derivative</b> class interface functions   | 192     |
| A.4.3 <b>Equation</b> class interface functions   | 192     |
| A.4.4 <b>Equation_Set</b> and <b>Dynamic_Set</b><br>class interface functions               | 193     |
| A.5 Physical Property Classes   | 195     |
| A.5.1 <b>Component</b> class interface functions  | 195     |
| A.5.2 <b>User_Component</b> class interface functions                                       | 196     |
| A.5.3 <b>Component_Set</b> class interface functions  | 196     |
| A.5.4 <b>General_Component_Mixture</b> class<br>interface functions                         | 197     |
| A.5.5 <b>Ideal_VLE</b> class interface functions  | 200     |
| A.6 <b>Mathtool</b> class interface functions   | 200     |
| <br><b>Appendix B: Flash Class Member Functions</b>   | <br>202 |
| B.1 Constructor   | 202     |
| B.2 Port Setup  | 205     |

|   |            |
|---|------------|
| B.3 Connection Functions                                  | 205        |
| <b>Appendix C: Tennessee Eastman Unit Models</b>          | <b>207</b> |
| C.1 Mixer Model   | 207        |
| C.2 Reactor Model   | 208        |
| C.3 Separator Model                                       | 210        |
| C.4 Stripper Model  | 211        |
| C.5 Nomenclature  | 212        |
| <b>Appendix D: Tennessee Eastman Flowsheet Definition</b> | <b>214</b> |
| <b>Appendix E: Fermentation Model Parameters</b>          | <b>221</b> |



# INDEX TO FIGURES

|   |     |
|---|-----|
| Figure 2.1: Simple layout of Tennessee Eastman Process.                     | 36  |
| Figure 2.2: Basic connection example.                                       | 37  |
| Figure 2.3: Stream class hierarchy.   | 38  |
| Figure 2.4: Port class hierarchy.   | 38  |
| Figure 2.5: Simple draining tank.   | 41  |
| Figure 2.6: Variable and Equation_Set class hierarchies.                    | 43  |
| Figure 2.7: Flow restriction valve between two tanks.                       | 44  |
| Figure 2.8: System class hierarchy.   | 47  |
| Figure 2.9: A flowsheet and its connected System-based tree.                | 49  |
| Figure 2.10: Connected mathematical tree of flowsheet in Figure 2.9.        | 50  |
| Figure 2.11: Virtual and polymorphic model functions.                       | 53  |
| Figure 2.12: Multiple inheritance numerical method class structure example. | 58  |
| Figure 2.13: Mathematical inheritance tree.                                 | 59  |
| Figure 2.14: Combined System/Mathtool class hierarchy.                      | 60  |
| Figure 2.15: Physical property class hierarchies.                           | 65  |
| Figure 3.1: Multiple access of Vector objects.                              | 71  |
| Figure 3.2: System-class steady-state analysis algorithm.                   | 74  |
| Figure 3.3: System-class steady-state collection/building algorithm.        | 75  |
| Figure 3.4: Combined System/Mathtool class hierarchy.                       | 77  |
| Figure 3.5: Port class hierarchy.   | 80  |
| Figure 3.6: Stream class hierarchy.   | 81  |
| Figure 3.7: Variable class hierarchy.                                       | 83  |
| Figure 3.8: Equation_Set building algorithm.                                | 85  |
| Figure 3.9: Dynamic_Set building algorithm.                                 | 86  |
| Figure 3.10: Tank volume balance equation tree.                             | 88  |
| Figure 3.11: Physical property class hierarchies.                           | 89  |
| Figure 3.12: Mathematical inheritance tree.                                 | 91  |
| Figure 4.1: Cylindrical liquid mixing tank.                                 | 96  |
| Figure 4.2: Flash vessel.   | 126 |
| Figure 4.3 Simulation instruction sequence.                                 | 131 |

|   |     |
|---|-----|
| Figure 5.1: Cavett Process.   | 136 |
| Figure 5.2: Tennessee Eastman Process.  | 142 |
| Figure 5.3: Reactor pressure response.<br>Reactor pressure setpoint change from<br>2806 to 2746 kPa absolute.         | 147 |
| Figure 5.4: Separator level response.<br>Reactor pressure setpoint change from<br>2806 to 2746 kPa absolute.          | 147 |
| Figure 5.5: Modified Luyben Control Scheme for Tennessee<br>Eastman Problem.  | 149 |
| Figure 5.6: Tennessee Eastman response to 15%<br>decrease in production rate.   | 153 |
| Figure 5.7: Tennessee Eastman response to product G:H mass ratio<br>setpoint change from 50:50 to 40:60.              | 154 |
| Figure 5.8: Tennessee Eastman response to reactor pressure setpoint<br>change from 2806 kPa. abs. to 2746 kPa. abs.   | 155 |
| Figure 5.9: Tennessee Eastman response to purge composition setpoint<br>change from 13.82 mole % B to 15.82 mole % B. | 156 |
| Figure 5.10: Fermenter control system diagram.  | 162 |
| Figure 5.11: On-off glucose control simulation.   | 163 |
| Figure 5.12: Full glucose control simulation. Setpoint 0.01 g /L.   | 164 |
| Figure 5.13: Full glucose control simulation. Setpoint 0.001 g /L.  | 165 |

# INDEX TO TABLES

|            |   |     |
|------------|---|-----|
| Table 4.1: | Composition and duty for vapour-liquid flash calculation.                     | 134 |
| Table 5.1: | Cavett feed and product stream compositions.                                  | 138 |
| Table 5.2: | Cavett flash specifications.  | 139 |
| Table 5.3: | Iterations and solution time to convergence<br>for the Cavett rating problem. | 140 |
| Table 5.4: | Iterations and solution time to convergence<br>for the Cavett design problem. | 140 |

# CHAPTER 1



## Introduction and Literature Review

Simulation may be defined as *reproducing the behaviour of a physical process by artificial means*. In a chemical engineering context it can be considered as the use of a computer to mathematically model and solve steady-state and dynamic process flowsheets. Chemical process simulation commenced in the late fifties when computer technology had advanced to the point where moderately complex programs were possible. Traditionally, process simulations and simulators have been coded in procedure-oriented languages such as Fortran. Simulations were originally limited to simple process systems or unit operations. These were coded on a problem-specific basis. The focus was on simulation of a chemical process from the corresponding unit model equations. This approach was still employed in the eighties (Shacham 1985) and can be a fast, useful tool for small problems. As computing power and languages developed through the sixties, simulation packages were designed that provided a variety of unit operations that could be linked together to simulate different process flowsheets. This is the type of simulation environment we are familiar with today.

### 1.1 Simulation Techniques

For a process flowsheet three basic methods can be used to find a solution. These are the sequential-modular, equation-oriented (simultaneous) and the parallel-modular methods. All are applicable to both dynamic and steady-state simulation, although the definition of parallel-modular as applied to dynamic simulation is unclear (Hillestad and Hertzberg 1986) and will not be discussed further. The methods are summarised below.

#### 1.1.1 Sequential-Modular

This was the initial method used for simulation of processes with recycle streams. The computational demands of the method were reasonably low. This was a significant factor for early simulations because a powerful digital computer in the early seventies only possessed 32kB of RAM and a 1 MB disk drive. Steady-state sequential-modular simulations proceed on a unit-by-unit basis around the flowsheet and convergence blocks

adjust stream variables until convergence is achieved. Design problems can be difficult to specify, although this is dependent on the unit model form. However, if units are modelled with simultaneous equations, design calculations are simple. Different locations and number of convergence blocks can dramatically affect simulation efficiency for a given flowsheet. Initial estimates are required for convergence block output streams to commence calculations.

Dynamic sequential-modular simulations have unit input variables approximated by polynomial interpolation for each time interval, with separate integrators applied to each unit module (Hillestad and Hertzberg 1988). This dynamic method sometimes copes poorly with strongly coupled systems (Fletcher and Ogbonda 1988). Steady-state sequential-modular simulation requires relatively few computing resources. DYFLO (Franks 1972) and DYN SYL (Patterson and Rozsa 1980) are examples of dynamic sequential-modular simulators. DYN SYL was originally developed for simulation of nuclear fuel processing systems and was coded in Fortran. DYN SYL included an option for fully equation-oriented simulation.

Sequential-modular methodology in its various forms has dominated commercially available process simulators until recently. Two notable steady-state simulators of this type are ASPEN (Evans *et al.* 1979) and PROCESS. ASPEN was originally developed at Massachusetts Institute of Technology and is programmed in Fortran. It employs a separate input file language for flowsheet description. In recent years a graphical interface (called ModelMaker) has been developed that translates a visual description of flowsheets into the input file language. ASPEN can also simulate multi-phase streams and solids processing. The simulator incorporates a large physical properties database.

### 1.1.2 Equation-Oriented

The equation-oriented approach considers the flowsheet as a whole and solves the set of flowsheet equations simultaneously using a Newton or quasi-Newton algorithm. This method is mathematically superior to other flowsheeting methods because the numerical solution is independent of the problem formulation. Equation-oriented dynamic simulation generally requires the solution of a set of algebraic equations with a set of differential equations. The two sets can be solved separately at each time step, or using polynomial

interpolation with the state or derivative values the differential equations can be solved simultaneously with the algebraic equations.

The computing resources required significantly exceed those for an equivalent sequential-modular simulation. The versatility of a system where the problem formulation is independent of the solution is offset by the effort required to provide a non-singular problem specification. Steady-state simulation requires a reasonable initial estimate for all solution variables to start the calculations. Convergence problems can occur with poor initial estimates or if the solution method encounters a 'difficult' solution region. Equation-analysis tools can assist with ensuring a non-singular problem. Variable estimates can be provided by a sequential-modular initialisation although this facility is extremely rare in equation-oriented simulators. Dynamic simulation requires consistent initial conditions for state and algebraic variables. These are often provided by the steady-state solution of the dynamic flowsheet.

SPEEDUP (Perkins and Sargent 1982; Pantiledes 1988) and QUASILIN (Hutchison *et al.* 1986a,b; Smith and Morton 1988) are equation-oriented simulators capable of steady-state and dynamic simulation. SPEEDUP has an input language for model definition and a symbolic manipulation tool for determining partial derivatives. The simulator executive is implemented in Pascal and the numerical routines are in Fortran. From the flowsheet definition the executive program translates the input file into Fortran code. SPEEDUP also has a database for file management. Physical properties are calculated as separate procedures to the simulation and not generally included in the equation set.

QUASILIN is similar to SPEEDUP in many respects, although the package is implemented completely in Fortran. Some thermodynamic properties can be included in the equation set although the majority are implemented as utility procedures through the physical property interface of the simulator. DIVA (Holl *et al.* 1988; Kröner *et al.* 1990a) is another equation-oriented dynamic simulator with similar features. The numerical methods and basic executive in DIVA are implemented in Fortran and the model building facility and user interface are developed in an expert system tool called KEE (Knowledge Engineering Environment). Physical properties are included in the flowsheet equation set.

### 1.1.3 Parallel-Modular

Steady-state parallel-modular simulation has three definitions. Firstly, there is a two-tier approach in which sequential-modular unit models are converged in turn with a linearised flowsheet equation set. The linearised set is obtained from perturbations of the sequential models. This method was developed to try and take advantage of the versatility of equation-oriented flowsheeting while utilising sequential-modular models. The simulator MASTEP (Timár *et al.* 1984) employs this method. MASTEP is coded in Fortran.

Secondly, sequential-modular outputs can be compared with equation-oriented outputs. This is known as a “tear every stream” approach and requires considerable computing resources. A third approach is a condensed version of the previous one and can equally be interpreted as sequential-modular. The tear stream sets for a sequential-modular simulation are considered as simultaneous equations. Each loop around a flowsheet becomes an extended set of nonlinear equations. The main advantage of this particular method is that interaction between tear variables is accounted for and the simultaneous solver can provide a better solution direction. The ASPEN simulator has this method as an option. Biegler (1983) presents a comprehensive review of parallel-modular flowsheet solution.

## **1.2 Object-Oriented Process Simulation**

Over the past few years, object-oriented programming techniques have been adopted for developing a variety of computer software. Object-orientation originated with the SIMULA language developed in the sixties (Dahl *et al.* 1968). The graphical interfaces of operating systems and computer-aided-drafting packages were among the first applications programmed using object-orientation. The basic philosophy of object-orientation matched well with the types of geometric objects encountered.

In object-oriented programming, the focus is on the data of the system under consideration. The data have directly associated behaviours or functions, *i.e.* the data can do things. This is in contrast to procedural programming languages where the emphasis is on functions that own and manipulate data. Object-oriented languages retain the manipulative capabilities of non-object-oriented languages with the additional capability to represent structure in a logical way. An object has been described as “...a chunk of structured knowledge.” (Stephanopoulos *et al.* 1987, p656).

The main principle behind object-orientation is the creation of user-defined *types*. These are often referred to as *classes*. An object is a specific instance of a class. A class can contain any data type that the programming language supports in addition to objects from classes defined by the programmer. The member objects contained in a class are called *attributes* of a class. A class can have its own behaviours, called member functions. A class is a definition of a data structure. An object contains the values or states of the data.

There are three basic concepts associated with classes and objects. They are *inheritance*, *polymorphism* and *encapsulation*. *Inheritance* means that a class can inherit attributes from another class. A derived class automatically contains the data structure and functionality of the class it inherits from. The inherited class is called a *parent* class. A derived class is called a *child* of the inherited class. Considering a computer-aided-drafting package, there could be a basic class of **Shape**. Specific shape classes, such as **Square**, **Triangle** and **Straight\_Line** could inherit a set of general shape attributes and behaviours from the **Shape** class. In some object-oriented languages (such as C++), a class can also be derived from several parents at one level. This is called *multiple inheritance*.

*Polymorphism* means that the same basic behaviour can be implemented in different ways. The **Square**, **Triangle** and **Straight\_Line** classes all need to know how to draw themselves on the screen. This may be accomplished by having a basic *draw()* function in each class. The name of the function or behaviour is the same for each class, but the method used to draw them would be different.

*Encapsulation* means that an object's data may be protected by permitting modification of the data only through operations (member functions) owned by that object. For data manipulation the class definition must provide the necessary operations and functionality. A sub-set of operations provides a controlled interface to the outside world. As well, they define how the data can be viewed externally. Encapsulation promotes data hiding by ensuring that data is accessed in a formally defined way. There are clearly varying levels of encapsulation that may be enforced in the class design, from completely loose, unprotected access to very strict control.



These concepts are readily extended to process simulation. The familiar entities such as streams, process units and components can be considered as basic classes. More specific classes can be derived from these. Member functions can be used for polymorphic unit models, solution methods and data transfer, *etc.*

### 1.2.1 Object-Oriented Simulation

The potential advantages of object-orientation for engineering application are well-documented (Lee and Arora 1991; Motard 1989). Westerberg and Benjamin (1985) have described the desirable characteristics of object-orientation as applied to process simulation. The authors expressed a desire for a building-block approach to process simulation. Process models should be constructed on a part-whole basis, with simple, tested constructions forming the basis of larger blocks or model parts which in turn form parts of the larger system and so on. The authors state:- "...the method of communication for building a model is through the use of a specially designed *nonprocedural* language...". Several other object-oriented language characteristics are discussed. Complete interaction with the simulation environment down to the individual unit model equations and variables is suggested to enhance debugging. A comprehensive database and expert-system based communication are also recommended.

Object-oriented simulation has followed two main paths of evolution. The first is the development of object-oriented simulation languages. The other is the development of object-oriented simulators and simulation environments.

### 1.2.2 Languages

Object-oriented simulation languages generally describe a process in terms of node and connector objects. The process can be interpreted in two ways (Bischak and Roberts 1991). The first interpretation is queue-based. The process is a network of queues and activities. Transactions pass through the network, where they wait in queues or are serviced by activities. The network is an object containing node and connector objects. Transaction objects are temporary because they enter and leave the various nodes. The relationship between objects and the process means that the objects flow through the process. Network simulation languages such as GPSS (Schriber 1991), SLAM (Pritsker 1986) and SIMAN (Pegden et al. 1990) employ this approach. This method of simulation is well suited to production line and schedule modelling, where discrete entities such as cars or television sets

move through a factory and are gradually constructed. The approach is not suited to chemical process simulation, primarily because flow of material through a pipe takes place continuously, rather than as discrete “lumps”. The use of network languages to simulate chemical plant operation and scheduling has been discussed in the literature (Morris 1992; Habchi and Deloule 1992).

The second interpretation is that process is an action and is a property of the node object, such as a member function. This is the normal implementation in most object-oriented programming languages. Languages such as MODEL.LA (Stephanopoulos et al. 1990a, 1990b), OMOLA (Nilsson 1993), gPROMS (Pantiledes and Barton 1992) and the modelling language in the ASCEND environment (Piela et al. 1990), have been developed and follow this interpretation. These languages and their major features will now be discussed with respect to chemical process modelling.

### *MODEL.LA*

The focus of MODEL.LA is on the automatic definition of a model from knowledge about the system. Model construction from system knowledge is recommended to accelerate the model-building process, reduce errors and separate model definition from model solution methods. The system knowledge includes modelling assumptions, simplifications and model purpose, which are unclear from a purely mathematical description. The language supports object-oriented principles and process representation through six modelling elements and eleven semantic relationships. These are described briefly below.

The modelling elements are:

- **Generic-Unit** :- A bounded system. The system can be a single unit, a group of connected units or a whole plant flowsheet.
- **Port** :- The interface between Generic-Units and the outside world. A Port is used to transfer information between Generic-Units.
- **Streams** :- These are connectors between Ports that enable Generic-Units to be connected together.
- **Modeling-Scope** :- A set of declarative relationships that apply to all the aspects of the model. It encapsulates the assumptions and relationships in a given model.

- **Constraint** :- Each relationship inside a Modeling-Scope is declaratively described by a Constraint. A Constraint is an unsolved relation among quantities, for example an equation.
- **Generic-Variable** :- A more complex version of a solution variable. It is used as a building block for describing modelling relationships.

Each modelling element is derived into more specific classes.

The semantic relationships are:

1. **Is-a** :- Denotes inheritance from a parent class.
2. **Is-a-member-of** :- Denotes an instance (object) of a class. The “member” terminology here refers to an instance declaration, as opposed to an attribute of a class in the earlier discussion.
3. **Is-composed-of** :- Describes the relationship between an object and the objects it contains. (Objects of a different type in the same context. See below, “Is-disaggregated-in”).
4. **Is-part-of** :- Reverse of the above. Describes the relationship between an object and the object that contains it.
5. **Is-attached-to** :- Defines connection from a Port to a Stream, or a Port to a Generic-Unit.
6. **Is-connected-by** :- Reverse of the above. Defines connection from a Generic-Unit to a Port, or a Stream to a Port.
7. **Is-described-by** :- Defines a link from a Generic-Unit or Port to a Generic-Variable or Constraint.
8. **Is-describing** :- Reverse of the above. Defines a link from a Generic-Variable or Constraint to a Generic-Unit or Port.
9. **Is-disaggregated-in** :- Breaks down groups of objects of the same type into more manageable pieces. (Objects of the same type in different contexts. See above, “Is-composed-of” ).
10. **Is-abstracting** :- Reverse of the above. Groups objects of the same type together.
11. **Is-characterized-as** :- Establishes a relationship between a modelling object and an attribute of its description.

The relationships (3,4), (5,6), (7,8) and (9,10) are symmetric in that invoking one member of a pair implies the other.

A process model is represented graphically. The graphical description is generated by algorithms that operate on the context (description and assumptions) of the model. The model is developed in applications that run on top of the expert-system tool KEE. This is in contrast to the majority of languages where models are developed in the language itself. MODEL.LA employs phenomena-based descriptions to generate the mathematical model. For example, by defining part of a model as “MASS-LUMPED-BALANCE”, the mathematical relations are automatically generated in MODEL.LA’s mathematical language. MODEL.LA supports equation-oriented modelling but incorporates no numerical methods, it is a model development and definition platform. The language is incorporated into the simulation environment DESIGN-KIT (Stephanopoulos 1987 *op.cit.*). DESIGN-KIT is discussed in section 2.2.2, Object-Oriented Simulation Environments.

### *OMOLA*

OMOLA is a general object-oriented data representation language. It is designed for general modelling purposes. It has four predefined classes, three of which are parents for other predefined classes. They are described below:

- **Model** :- The base parent class for all user-defined models.
- **Terminal** :- The base parent class for all model interaction classes. Similar to the Port class in MODEL.LA.
- **Parameter** :- A user-specified constant value, for example a tank surface area.
- **Variable** :- The base parent class for mathematical variables.

The general syntax for a user-defined class is :

```
{name} ISA {name of parent class} WITH
      {class body}
END;
```

The class body contains the attributes of the class. The attributes can include class definitions, variables and equations. OMOLA does not support member functions or operations in the conventional object-oriented sense. The language employs *equations* which perform the same function. OMOLA employs only two semantic relationships but each works in two ways. The relationships are:

1. **ISA** :- Denotes inheritance from a parent class, as indicated above. It also denotes an instance of a class.
2. **WITH...END** :- Used to define the set of attributes in a class definition. It is also used to initialise some attributes of the objects that a class owns.

OMOLA class definitions include sections for various modelling entities, for example Terminals, Submodels, Parameters, Variables, Equations and Connections. There is no structure for the definition of connecting streams, connections are made directly via Terminal-hierarchy objects with an "AT" relationship. An example of the definition of a simple tank is given below. The ' after the mass variable denotes a time derivative.

```
TankModel ISA Model WITH
terminals:
    In          ISA PipeInTerminal;
    Out         ISA PipeOutTerminal;
    Level       ISA SimpleTerminal;
parameters:
    Density     ISA Parameter;
    TankArea    ISA Parameter;
variable:
    mass ISA Variable;
equations:
    mass' = Density*(In.Flow-Out.Flow);
    mass = Density*TankArea*Level;
END;
```

OMOLA does not offer some programming language features such as arrays and looping. OMOLA is part of the OMOLA Simulation Environment, called OMSIM (Mattsson et al. 1993). Like MODEL.LA, equation-oriented models are supported and numerical methods are provided by a companion simulation environment, called OMSIM. OMSIM is discussed in section 2.2.2, Object-Oriented Simulation Environments.

### *ASCEND modelling language*

The modelling language in the ASCEND environment focuses on the representation of nonlinear algebraic systems. The primary goal of ASCEND is to provide an equation-oriented modelling environment. The environment consists of the modelling language and a set of tools for manipulating the structures created in ASCEND. A variety of simultaneous equation solvers are included. ASCEND is similar to OMOLA in many respects. The language contains three predefined classes for model building:

- **Model** :- The base parent class for user-defined models.
- **Atom** :- Denotes a physical quantity, *e.g.* length.
- **Type** :- Elementary language data type, *e.g.* int.

Object-oriented semantic relationships are provided with the following statements:

1. **REFINES** :- Denotes inheritance from a parent class.
2. **IS\_A** :- Denotes an instance (object) of a class.
3. **IS\_REFINED\_TO** :- Changes the type associated with a previously defined object.
4. **ARE\_ALIKE** :- Operates on a group of objects and forces them to be of the same type. The type is the most derived class of the group of objects.
5. **ARE\_THE\_SAME** :- Takes 4) a stage further and merges the group of objects into one instance or object.

A specific instance (object) can be further declared as UNIVERSAL. This creates a single instance that can be referenced throughout a model. It operates the same way as the ARE\_THE\_SAME relationship. The ASCEND modelling language does not contain a Port/Stream data structure for connectivity. Connections are defined with the

ARE\_THE\_SAME relationship. Dynamic simulation is not directly supported. Integration methods can be programmed in as Model-hierarchy classes. The tank example given below assumes the Atoms for `vol_flowrate`, `length`, `density`, `mass` and `mass_flowrate` are already defined and that an integration method has been programmed.

```

MODEL TankModel REFINES model;
    no_of_states := 1;
    in_flow, out_flow IS_A vol_flowrate;
    level             IS_A length;
    rho              IS_A density;
    m                IS_A mass;
    dm_dt            IS_A mass_flowrate;
    area             IS_A generic_real;
    area.fixed       := true;

    y_prime[1], dm_dt ARE_THE_SAME;
    y[1], m           ARE_THE_SAME;
    t                 IS_REFINED_TO time;

    dm_dt = rho*(in_flow - out_flow);
    m     = area*rho*level;

END TankModel;

```

In this example the integrator has a number of states which must be set prior to solution. This is done with the assignment `no_of_states := 1;`. The tank model merges the time derivative object `dm_dt` with the `y_prime[1]` object of the integrator, and the state variable `y[1]` is merged with the mass `m`.

One interesting aspect of ASCEND is that it promotes open data structures, to the point where the philosophy is almost anti-encapsulation. In traditional software engineering and object-orientation, the principle of encapsulation or information hiding is standard for

protecting the internal data of an object or software module. The ASCEND group contend that information hiding impedes the development and debugging process by preventing direct access to structure and attributes. They suggest that in an extendable system the user should be able to work at whatever level of abstraction they consider appropriate and as such all elements of the system should be accessible.

### *gPROMS*

gPROMS is an object-oriented extension of the modelling language present in the SPEEDUP simulator. The language is designed to model combined discrete and continuous processes. The syntax of gPROMS is very similar to the preceding languages. A variety of classes is available for constructing unit models. Predefined class hierarchies exist for streams, physical quantities and elementary data types *etc.* Object-orientation is derived from the following two relationships:

1. **INHERITS** :- Denotes inheritance from a parent class.
2. **AS** :- Denotes an instance of a class.

Connectivity is defined with an “IS” relationship. The tank example is given in gPROMS below. The \$ symbol denotes a time derivative.

**MODEL TankModel**

**PARAMETER**

**TankArea AS REAL**

**VARIABLE**

**Flow\_In, Flow\_Out AS Volume\_Flowrate**

**m AS Mass**

**rho AS Density**

**h AS Length**

**STREAM**

**Inlet :Flow\_In AS Mainstream**



Outlet                   :Flow\_Out AS    Mainstream

EQUATION

```
$m = rho*(Flow_In - Flow_Out) ;  
m = h*TankArea*rho ;
```

END

gPROMS is being further developed to model distributed parameter processes (Oh and Pantiledes 1994) and has a facility for translating its models into other languages, such as ANSI C.

### 1.2.3 Object-Oriented Simulation Environments

Three broad classifications are possible for simulation environments. The first includes the familiar simulators that provide a library of existing process models that the user can connect together to create flowsheets. Their capacity for user-defined model construction is limited. The second is complete modelling and simulation environments that incorporate a library of process models and in addition have a facility and/or a language for user-defined models. The third classification covers interfaces to existing software that are designed to bring together different useful aspects of several packages. Object-oriented approaches to the three classifications have been developed. Some of these developments are discussed below.

#### *IOWA STATE UNIVERSITY*

A prototype steady-state object-oriented simulator was developed at Iowa State University (Gadjaru 1992, Lau 1992). The project aim was to investigate process simulation and its application to object-oriented process integration. The simulator is coded in C++. The simulator can employ sequential-modular simulation to initialise an equation-oriented simulation. Several sequential and equation-oriented solution methods are available. It is not capable of dynamic simulation. Flowsheet definition is based on text input with C++.

Object-orientation is employed at many levels in this work; almost everything that the user manipulates is an object. The equations in a model are represented as sets of binary tree objects. The elementary data structures are also implemented as class hierarchies, for

example, matrices. Mathematical operations on matrices are accomplished by overloaded operators, which enables matrix addition, say, to be written in the C++ code as  $C = A + B$ . Tools are provided for sequential-modular tear set selection, equation analysis and partitioning.

Model extension is considered a desirable and necessary feature, although a model construction facility is not described. The authors stress that extension should be at the object level whenever possible. Thus, some objects should be user-modifiable and the system still includes extension at the class level for more experienced users/programmers. Dynamic simulation facilities are not provided.

### *DESIGN-KIT*

DESIGN-KIT is a knowledge-based support environment for process engineering. The emphasis is on knowledge-based analysis rather than pure simulation. It incorporates MODEL.LA as the associated modelling facility. The objective was to create a homogeneous package for all aspects of process engineering - flowsheet synthesis, control synthesis, scheduling, planning, simulation and equipment costing and design. DESIGN-KIT is developed in CommonLISP and KEE. CommonLISP is employed for the user interface and KEE is employed for process knowledge and model description. The package supports steady-state equation-oriented simulation and has a variety of tools, including equation analysis, order-of-magnitude reasoning and a rule interpreter. The environment incorporates a graphical interface for the various activities. Dynamic simulation facilities are not provided.

### *OMSIM*

OMSIM is the simulation environment for OMOLA. OMSIM is implemented in C++ but user-defined model classes must be written in OMOLA. OMSIM contains facilities for model development either as direct OMOLA text or a graphical interface and class browser for mouse-based model construction. It also provides the software for simulating the developed models and a database for model storage. Simulation of a model is a two-step process. First, the model is checked to see if it is lexically and syntactically correct and that all types and connections are valid. This is similar to compilation in a normal programming language. Second, the resulting mathematical structure is analysed. The

analysis checks for singularity, attempts to resolve high-index problems in the dynamic equations and performs partitioning and symbolic manipulation. Simulation and modelling are at the class level. A model forms part of the definition of a new class, as opposed to dynamically connecting everything at the object or instance level.

A wide variety of numerical methods are provided for integration of ordinary-differential- and differential-algebraic-equations. OMSIM does not have steady-state numerical methods. The interface can be interactive during a simulation and results are available graphically on-the-run.

### *MODELER*

MODELER is an object-oriented environment for the modelling of physico-chemical-biological systems (Lee 1991b). The environment is implemented in MODULA-2. The building-blocks of MODELER are structures based on Newtonian physics and axiomatic thermodynamics. The structures are used to form conservation relations in mass, energy and momentum. Five primitives are defined :

- **Phase** :- A region within a system with homogeneous properties.
- **Physical Lumped System** :- A region of space, enclosed by a boundary and specified by a given set of state variables.
- **Chemical System** :- A system containing stoichiometric and kinetic information about a chemical system associated with a physical lumped system.
- **Physical Property System** :- A system containing numerical values of and/or calculation methods for physical properties.
- **Information System** :- A system that contains a mechanism for transforming information (e.g. a controller).

A system model is built from the primitives and connection streams with a graphical interface. The system is represented as a block diagram. Mass and energy balances are constructed automatically from the primitives and connectivity in a model. MODELER does not provide numerical methods.

### *ÉPÉE*

ÉPÉE (Ballinger *et al.* 1994) is an object-oriented interface system for process engineering. The goal of the project is to share data amongst process engineering software packages. It provides a set of common process-engineering objects that are user-extensible, for example **process**, **stream** or **component**. The software packages are considered to be methods of ÉPÉE. The packages available are the steady-state ASPEN simulator, the PPDS physical property system (N.E.L. 1982) and a heat integration package called CHiPS (Fraga *et al.* 1991).

### *VeDa*

VeDa is an object-oriented process modelling paradigm (Marquardt 1993). It is based on 'Substantial Modelling Objects'. The main objects are described as:

- **Devices** :- These are the things in a process, similar to the Generic-Unit or Models described earlier.
- **Elementary-Devices** :- A device that is non-decomposable. Similar in some ways to the Type class in ASCEND.
- **Connections** :- Objects that connect Elementary-Devices together.
- **Composite-Devices** :- An aggregation of Elementary-Devices joined together with Connections. Composite-Devices can be joined together to form more complex Composite-Devices.

The object-oriented data model is similar to the principles of other object-oriented simulation languages. It employs user-defined types with multiple inheritance, attributes and methods to define systems. Knowledge-based descriptions are implemented and a database is required for system management. A translator to turn VeDa descriptions into code compatible with the DIVA simulator is under development.

### *KBMoSS*

KBMoSS is a recent development (Vázquez-Román *et al.* 1996). It is a knowledge-based modelling support system. Models can be generated automatically from knowledge-based descriptions and the user can modify the generated models or develop one from first

principles without the knowledge-base. The environment and modelling procedure are based on five characteristics:

- A modelling procedure is a knowledge-based exploration task. Therefore, tools for exploration of alternatives, refinement and reasoning are necessary.
- Model development is evolutionary. Evolution represents the progressive improvement of a model.
- Cooperation (within the system) enhances the sharing of data and knowledge.
- The support system should integrate all the tools and information required for modelling.
- Automation facilitates analysis and promotes consistency.

KBMoSS is implemented in CommonLISP and can write final unit models in gPROMS. It does not provide numerical techniques for simulation.

#### 1.2.4 Summary of Object-oriented Simulation

The languages and their associated examples follow a similar approach. They are all part of a larger simulation environment but are implemented differently. ASCEND and OMOLA support only single inheritance. MODEL.LA and the VeDa data model support multiple inheritance. Single inheritance can achieve the same structural and functional goals as multiple inheritance, but it produces a longer class hierarchy. ASCEND promotes open access to all the elements of a data structure with no encapsulation.

MODEL.LA, MODELER and to a certain extent VeDa employ process knowledge to construct models from a set of relationships and assumptions. The other languages require a textual description of the class structure and modelling equations. KBMoSS supports both knowledge and textual description. The objective of knowledge-based synthesis is to provide complete documentation of the modelling process, increase consistency and reduce errors. The use of database facilities for maintenance and model development is common. The level of sophistication varies, from basic file saving in MODELER to the knowledge-based applications in MODEL.LA and DESIGN-KIT.

Some of the languages provide standard looping and conditional structures similar to those in procedural languages (e.g. FOR and IF~~ELSE~~ constructs) and arrays. In addition gPROMS includes a library of commands for discrete-event processing and distributed-parameter processes.

The development of object-oriented interfaces to other software is interesting. It can exploit the large programming effort expended on existing packages and present the user with a single consistent simulation platform. A potential major disadvantage of such an interface is that it could require many different interpreters and translators to provide user-extension to the software it drives.

### **1.3 Biochemical Process Simulation**

Biochemical process simulation is a relatively immature field when compared to process simulation in conventional chemical processing (Villadsen 1989; Petrides and Cooney 1993). The main reasons are summarised below.

The primary reason is system complexity. Biological systems are complex by nature and therefore are difficult to describe qualitatively and mathematically. Some aspects of the system under consideration could be irrelevant. In addition, biochemical researchers often do not have a process-engineering background and have been reluctant to apply mathematical modelling principles to biochemical processes until recently.

The products from biochemical processes are often difficult to characterise because of their complicated chemical structure. This makes the calculation of physical properties difficult. Many bioprocess unit operations are poorly understood and models are hard to develop. Most bioprocesses are batch or semi-continuous. These require dynamic models and simulation facilities.

Another reason is not stated in the above articles. It concerns the financial difference between the high-volume commodity products of the traditional process industries as opposed to the low-volume specialised products of biochemical plants. A great proportion of the world's economy has been dependent on petroleum products for many years. The corresponding competition provides enormous financial incentive for development of highly efficient

processes, in which process simulation plays a significant role. Low-volume, high-added-value bioproducts are not exposed to the same levels of competition and corresponding incentive for optimisation with process simulation. For state-of-the-art bioproducts, patent protection further reduces the financial incentive for modelling. In those bioprocesses where process scale and competition has significant economic effect, the level of understanding of the bioprocess is considerably greater and simulation has had much wider application. Two good examples are ethanol production from yeast and the production of penicillin.

A number of software packages have been developed for bioprocess simulation. Several different approaches have been taken and are discussed below.

#### *ASPEN BioProcess Simulator (BPS)*

BPS is an extension of the ASPEN process simulator (Evans 1988; Petrides *et al.* 1989). A large variety of biochemical unit operations have been added to the existing simulator. It is not capable of dynamic simulation. Instead, a time-average is applied to batch operation modules. This produces a pseudo-continuous operation that can then be solved with the steady-state simulator. BPS does not provide a modelling facility.

#### *BioPro Designer*

BioPro Designer is an interactive synthesis/analysis program (Petrides 1994). The synthesis tools are knowledge-based and are used to create flowsheet topologies. The user inputs details such as product properties and micro-organism type. The resultant flowsheet can then be simulated with the analysis component of the program. The analysis component can perform material and energy balances and economic evaluation. The knowledge-based tools are implemented with an object-oriented environment named Nexpert and the analysis tools are programmed in ANSI C. The program incorporates a graphical interface. BioPro Designer does not provide a modelling facility.

#### *CAMBIO*

CAMBIO is a knowledge-based environment for modelling and simulation of bioprocesses (Farza and Chérury 1991). It exploits prior knowledge to aid in flowsheet synthesis and model construction. Models can be taken from the supplied library or modified by the user. The material balances are generated automatically. Models are constructed from

basic elements such as substrates, biomass, enzymes and reactions, *etc.* with a graphical interface. The final system is modelled as a set of differential-algebraic equations (DAEs). The DAEs are then compiled as a subroutine and solved with the DC03AD integration package in the HARWELL code library. CAMBIO is written in the Pascal language.

### *SIMBIOS*

SIMBIOS is a steady-state simultaneous-modular bioprocess simulator (Simon *et al.* 1994). The simultaneous-modular technique employed is the two-tier method where unit outputs are calculated and then used to linearise a flowsheet model. The simulator contains an interface to a large physical property system. Dynamic simulation facilities are not provided.

### *BioSep Designer*

BioSep Designer is a knowledge-based flowsheet synthesiser focussing on protein separation systems (Siletti 1990). It determines an optimum flowsheet from a set of input data. A database is provided for protein properties. BioSep Designer is programmed in CommonLISP.

### *gPROMS*

The language gPROMS, discussed earlier, has been employed for bioprocess simulation (Lu *et al.* 1994). The production of alcohol dehydrogenase from yeast cells was examined. Five unit operations were considered in the flowsheet: fermentation, centrifugation, homogenisation, debris removal and product precipitation. The main advantage cited for gPROMS was that it is capable of simulating dynamic discrete events and could therefore be used for each batch process in the flowsheet.

## 1.3.1 Summary of Bioprocess Simulation

There are few bioprocess simulation packages compared to traditional process simulators. The extension of existing simulators to bioprocesses (e.g. ASPEN BPS) provides a robust platform for simulation but restricts the application to simulation methods supported by the base system. ASPEN is restricted to steady-state sequential-modular simulation. Of the biochemical simulators discussed, only BioPro Designer and CAMBIO can perform dynamic simulation. CAMBIO supports model extension. BioSep Designer and BioPro Designer



employ process knowledge for flowsheet synthesis. BioSep Designer is purely a synthesis program and has no simulation facility.

#### **1.4. Physical Property Calculation**

Physical property calculation is an area of study comparable in size to that of process simulation. The combination of the enormous variety of calculation methods and the number of chemical components likely to be required by a process simulator makes the development of a comprehensive physical property package a daunting task. Specific property calculation packages or methods are not reviewed but a general discussion of the requirements for process simulation is presented. Calculation of properties for gases and liquids is discussed in Reid (1988).

It has been estimated that up to 80% of the computer time required for a simulation is consumed by physical property calculation (Westerberg 1979). Four principal capabilities were identified:

1. Supply estimates for several physical properties for several different components during the course of a simulation.
2. Provide the user with values for properties of interest during the simulation and after completion.
3. Allow user-defined property data.
4. Supply estimates where data is poor or unavailable.

The representation of a mixture of components and corresponding physical properties interact strongly in a simulator. Britt (1980) provides a detailed discussion in the context of steady-state simulation and process streams. Salient points include:

1. A stream can consist of many phases: solid, liquid and vapour. The source unit of the stream establishes the phase condition. Any combination and number of phases can occur, with the restriction of a single vapour phase.
2. Liquid and vapour phases can be assumed to be at equilibrium. Solid phases might or might not be at equilibrium.

3. The level of information associated with a stream can vary. If a stream consists of several phases, is a composition for each phase required, or just an average composition for the whole stream?
4. Some solid phase properties can be characterised in terms of pure component data similar to vapour-liquid equilibrium. Other solid properties could require further characterisation.
5. Some solid phases cannot be characterised in terms of pure component data. The material must be otherwise characterised. The process stream must cater for this alternative characterisation. Property calculation for the characterisation will be different.
6. The necessary characterisation is dictated by the requirements of the unit model.

Complications arise when unconventional components are considered. Bioprocesses are sources of unconventional components. How is a cell characterised? Generally a molecular formula for a cell species can be experimentally determined in terms of carbon, nitrogen, oxygen and hydrogen, but this indicates nothing else about the physico-chemical properties. One option is the definition of properties on a micro or macro level. Conventional components can often be characterised in terms of their structure and interactions on a molecular or micro level. Unconventional components could be characterised in terms of other attributes, for example size or bulk density which are properties that can be determined at the macro level. This is applicable to solids and slurry processing and bioprocesses.

The implementation of physical property methods can be approached in two ways. Physical property data can be calculated as a utility to the unit models. This is the case for the majority of process simulators. The assumption of vapour-liquid equilibrium in steady-state simulation is usually justified and means that property data can be calculated explicitly from process conditions.

## 1.5 Numerical Analysis Methods

Considerable research effort has been expended in the development and testing of numerical methods for process simulation. Comprehensive reviews are available (Seider and Brengel 1991; Shacham 1985; Sargent 1981). Numerical methods applicable to steady-state and dynamic simulation are discussed below. The reader is referred to the review articles above for further information. Methods for the solution of distributed-parameter systems are not discussed.

### 1.5.1 Nonlinear Algebraic Equations

Solution algorithms for nonlinear algebraic equations fall into two categories, explicit and implicit. Explicit solution methods require the variables of interest (output variables) in each equation to be available explicitly as a function of the other variables ( $\mathbf{x}$ ) and parameters ( $\mathbf{u}$ ) in the equation set, *i.e.*

$$\mathbf{x} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \quad (1.1)$$

Usually the explicit output variables are functions of themselves and an iterative substitution is required to converge the equations. This method was employed in early simulators and is the reason sequential-modular simulators were criticised for coping poorly with some problem specifications. If a problem specification required solution for a variable not in the explicit output set, a further iterative loop was required to solve for the variable.

Implicit methods solve a system of simultaneous equations by forcing a set of residual values to below a specified error tolerance. The model formulation is:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (1.2)$$

An explicit equation formulation can be made implicit by the transformation:

$$\mathbf{f}(\mathbf{x}) = \mathbf{g}(\mathbf{x}) - \mathbf{x} = \mathbf{0} \quad (1.3)$$

Solution of the system of equations follows an iterative procedure:

$$\mathbf{J}(\mathbf{x}_n) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}_n} \quad (1.4)$$

$$\mathbf{J}(\mathbf{x}_n) \Delta \mathbf{x} = -\mathbf{f}(\mathbf{x}_n) \quad (1.5)$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta \mathbf{x} \quad (1.6)$$

The Jacobian  $\mathbf{J}(\mathbf{x})$  can either be the true partial derivatives of  $\mathbf{f}(\mathbf{x})$  as in the exact Newton's method or an approximation, such as the update formula in Broyden's method (Broyden 1965). The partial derivatives can be calculated either by analytical differentiation of the equations, or by a numerical finite difference approximation.

Hybrid algorithms exist where the  $\Delta \mathbf{x}$  term is calculated as a weighted combination of the Newton direction given above and the direction of steepest descent. Powell's dogleg (Acton 1990) and Marquardt's method (Marquardt 1963) are examples.

Systems of equations in flowsheeting generally do not have every variable occurring in every equation. Often the systems are sparse and have only a few percent non-zero entries in the Jacobian matrix. Considerable savings in execution time are possible if numerical methods are employed that take advantage of sparsity in the solution for the  $\Delta \mathbf{x}$  term above. Reviews of sparse numerical techniques are available in Bogle and Perkins (1988) and Stadtherr and Wood (1984 a, b).

Another more recent development in numerical methods for nonlinear equations is the use of homotopy paths (Wayburn and Seader 1987). Homotopy methods are designed to promote convergence of a nonlinear system of equations from a poor initial estimate:

The objective of a homotopy method is to obtain the solution of the set of equations

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (1.2)$$

by the solving the homotopy

$$\mathbf{h}(\mathbf{x}, t) = \mathbf{0} \quad (1.7)$$

The homotopy  $\mathbf{h}(\mathbf{x}, t)$  defines a path  $\mathbf{x}(t)$ , where  $t$  is a mathematical parameter. Assume that  $\mathbf{h}(\mathbf{x}, 0) = \mathbf{0}$  has a known solution. Also assume that  $\mathbf{h}(\mathbf{x}, 1) = \mathbf{f}(\mathbf{x})$ . If  $\mathbf{x}^* = \mathbf{x}(1)$ , then  $\mathbf{x}^*$  is a solution of  $\mathbf{f}(\mathbf{x})$ . Therefore if a homotopy function  $\mathbf{h}(\mathbf{x}, t)$  exists and the path  $\mathbf{x}(t)$  can be followed from an initial estimate  $\mathbf{x}(0)$  to  $\mathbf{x}(1)$ , the solution to  $\mathbf{f}(\mathbf{x}) = 0$  can be found.

One method of solution for a homotopy is to transform the problem into a set of ordinary differential equations. Consider the function  $\mathbf{h}(\mathbf{x}, t)$ , with  $\mathbf{x} = \mathbf{x}(t)$ . Taking the derivative of  $\mathbf{h}(\mathbf{x}, t)$  with respect to  $t$ , we obtain:

$$\frac{d\mathbf{h}(\mathbf{x}(t), t)}{dt} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{d\mathbf{x}}{dt} + \frac{\partial \mathbf{h}}{\partial t} = 0 \quad (1.8)$$

A variety of choices exist for the homotopy. Consider the Newton homotopy, given by:

$$\mathbf{h}(\mathbf{x}, t) = \mathbf{f}(\mathbf{x}) - (1 - t)\mathbf{f}(\mathbf{x}_0) \quad (1.9)$$

where  $\mathbf{x}_0$  is the initial estimate for the system. The transformation to ordinary differential equations becomes:

$$\frac{d\mathbf{h}(\mathbf{x}(t), t)}{dt} = \frac{\delta \mathbf{f}}{\delta \mathbf{x}} \frac{d\mathbf{x}}{dt} + \mathbf{f}(\mathbf{x}_0) = 0 \quad (1.10)$$

This is an initial-value-problem to be integrated on  $t = [0, 1]$  with initial condition  $\mathbf{x}(0) = \mathbf{x}_0$ . The coefficient matrix of the derivatives is the Jacobian of the nonlinear system. Paloschi (1996) discusses the application of sparse methods to homotopy solution.

### 1.5.2 Integration Methods

Dynamic process simulation normally requires the integration of a set of ordinary differential equations (ODEs). A large number of ODE integration algorithms exist, such as explicit and implicit Runge-Kutta, Backward-Difference, Adams-Moulton, Burlisch-Stöer *etc.* Often with the set of ODEs there is an associated set of algebraic equations (AEs). The AEs are often nonlinear. The combined set of ODEs and AEs is referred to as a differential-algebraic-equation (DAE) set.

DAE sets can be solved in two ways:

- Solve the AEs with a simultaneous solver and then the ODEs with an integration algorithm.
- Solve the DAE set as a whole, converging AEs and ODEs at the same time.

It has been found that it is generally more efficient to solve the DAE set as a whole rather than AEs and ODEs separately (Marquardt 1991). The most widely-used DAE algorithm is Gear's Backward-Difference-Formulae (BDF) method (Gear 1971). The BDF method approximates the vector of derivatives in the system with polynomial backward differences of various orders. The simplest is the first-order backward difference formula, where Euler's approximation is applied to the derivatives as below:

$$\dot{\mathbf{x}}_{n+1} = \frac{d\mathbf{x}_{n+1}}{dt} = \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{t_{n+1} - t_n} \quad (1.11)$$

The DAE system can then be solved as a set of algebraic equations driven by an integration algorithm as:

$$\mathbf{F}(\dot{\mathbf{x}}_{n+1}, \mathbf{x}_{n+1}, t_{n+1}, \mathbf{u}) = 0 \quad (1.12)$$

where  $\mathbf{x}$  denotes the vector of state variables and  $\mathbf{u}$  denotes the vector of inputs or design specifications for the problem. This formulation has many advantages. The equation structure is preserved, which means that sparse matrix techniques can be readily applied. The solution method is applicable to DAE and ODE problems, systems with implicit derivatives and stiff systems of equations. Stiff equation systems arise frequently in chemical reaction

kinetics. Stiffness arises in systems with time constants differing by several orders of magnitude. This is not a strict mathematical definition of stiffness. Lambert (1991) presents a discussion on stiffness of linear and nonlinear DAE systems.

The solvability of a system of ODEs or DAEs can be characterised by a parameter called the index. There are a few definitions of the system index (Unger *et al.* 1995). Generally the differential index of a system is referred to. The differential index is defined as the minimum number of times the DAE system  $\mathbf{F}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{u}) = \mathbf{0}$  must be differentiated with respect to time in order to determine  $\dot{\mathbf{x}}$  as a continuous function of  $\mathbf{x}$  and  $\mathbf{u}$ . Systems with indexes of zero or one are easy to solve with standard ODE or DAE methods. A system of ODEs that are solvable has an index of zero. The addition of one or more algebraic equations raises the index to one. Higher index problems result when a state variable response is specified as a function of time and the required system input must be determined. The concept of system index is best illustrated with examples.

Consider a simple lumped-parameter mass balance of liquid inside a cylindrical tank. The tank has one input and one output stream. The mathematical model is :

$$\frac{dM}{dt} = F_{in} - F_{out} \quad (1.13)$$

where  $M$  is the total mass of liquid in the tank and  $F_{in}$  and  $F_{out}$  are the mass flows in and out respectively. For specified  $F_{in}(t)$  and  $F_{out}(t)$ , the integration of the differential equation in  $M$  is an index zero problem, solvable by any ODE integration algorithm. Now consider calculating the height  $h(t)$  of liquid in the tank with an algebraic equation. The equation set becomes:

$$\frac{dM}{dt} = F_{in} - F_{out} \quad (1.14)$$

$$0 = \rho Ah(t) - M(t) \quad (1.15)$$

where  $\rho$  is the density of the liquid and  $A$  is the tank area. The addition of the algebraic equation raises the index to one. Any DAE algorithm, given valid initial conditions can solve the system. The index is raised to two if the height profile of the tank with time is specified

and the required input flow profile  $F_{in}(t)$  is required. The mass profile  $M(t)$  calculated from the  $h(t)$  function must be differentiated in order to determine  $F_{in}(t)$ .

Index reduction can be accomplished by careful modelling (Lefkopoulos and Stadtherr 1993), symbolic manipulation of the equations in the system (Chung and Westerberg 1990; Gear 1988) and structural analysis of the equations (Unger *et al.* 1995; Pantiledes 1988).

## **1.6 Conclusions and Project Scope**

The areas discussed offer considerable scope for research. The following conclusions are drawn from the discussion:-

- Systems that have a facility for model construction are more versatile than the traditional black-box process library in older simulators. In leading-edge technology, particularly biotechnology, process information remains proprietary. Off-the-shelf simulators are unlikely to contain models for novel unit operations. A modelling facility is practically a necessity in a modern simulation environment.
- Object-orientation assists model development, but object-orientation in itself is not sufficient to define a modelling methodology. Models require a logical and consistent structure. The concept of systems containing ports that can be connected to other systems is prominent. Complex systems should be modelled through decomposition into smaller component systems. This is often referred to as hierarchical decomposition and further promotes the object-oriented principles of software re-use and extension.
- Systems that can apply knowledge to the model development process have advantages, but modelling should not be exclusively a knowledge-based activity.
- Steady-state and dynamic simulation should be supported. Dynamic methods are required for control systems and biochemical process simulation.
- Equation-oriented unit models should be preferred, but not necessarily exclusively.
- In steady-state simulation, equation-oriented, sequential-modular and parallel-modular flowsheet calculations all have advantages. Ideally all three should be available in an integrated form. The equation-oriented approach is superior for dynamic simulation.



- There are many requirements for physical property calculation, depending on the process and simulation type. The physical property structure should be simple to allow the integration of different property methods into the system.
- Equation analysis and numerical tools are required in a complete simulation environment.

From the conclusions, the primary project objective was defined:

**To develop a basic object-oriented data structure and tools for the modelling and simulation of chemical and biochemical processes.**

The sub-objectives were defined as follows:

1. Provide steady-state and dynamic capabilities, with the ability to transition from steady-state to dynamic simulation.
2. Provide a variety of steady-state and dynamic solution methods.
3. Provide interchangeable steady-state simulation methods.
4. Provide a basic physical property and component data structure and methods.
5. Provide a reasonably simple structure and methodology for the definition of model classes and associated equations.

Some implementations and topics discussed in this chapter are considered outside the scope of the primary objective. A knowledge-based implementation cannot be developed until a validated mathematical structure exists. The numerical methods are based on the “traditional” Newton-based methods and Gear’s Backward Difference method. Homotopy methods and index analyses are unnecessary at this stage of the project’s development.

The design of a class structure based on the attributes of a process flowsheet and the project objectives is discussed in the next chapter.

# CHAPTER 2

## Simulator Development and Data Structure

In this chapter the development language is discussed briefly using a simple example. This is followed by the design of the basic class structure for the simulator. The design of the class structure results from an examination of the various characteristics of a process flowsheet and an analysis of the requirements for simulation of a flowsheet. Class names are printed in bold type.

### 2.1 Development Language

Several languages specifically developed for process modelling have been discussed. In this work, it was decided to exploit a highly refined and commercially available object-oriented language (namely C++) instead of developing another special-purpose modelling/simulation language. The use of an existing numerically-oriented commercial language provides a number of benefits:

1. Language structure. A programming language must be subject to rigorous testing and development for successful commercial application. This is extremely important for an object-oriented language that naturally encourages language extension. In a commercialised object-oriented language, debugging is restricted to the programmer's language extensions rather than the language itself, which would be the case if a new language was developed.
2. Consistent software platform. The final purpose of simulation and modelling languages is to define a numerical problem from a process structure. A simulation environment must support flexible and realistic modelling of processes while promoting efficient numerical analysis. Simulators have been described which employ different languages for modelling, numerical methods and the interface. This increases complexity. A more versatile simulation environment can be developed if all the facilities are coded in the same language.

3. Versatile development environment. Powerful compilation and debugging environments are available for commercial programming languages. The development of such environments is not a trivial task (consider the MODEL.LA/DESIGN-KIT system, (Stephanopoulos et. al. 1987, 1990 a,b)).
4. High portability. A language in common use offers high portability between different operating systems and computers.
5. Pre-defined data types, operators, looping and utility function libraries. Process simulation calculations require repetitive numerical calculations and logical decisions. Facilities for input, output and file manipulation are also required. These are provided by existing commercial languages.
6. Potential interfaces to existing software. Employing an existing commercial language enables access to a vast library of potential software, depending on the application. The development of libraries of numerical software is very common. Such libraries generally have simple functional interfaces to the developed code, enabling easy incorporation into a simulator's numerical library. Many language development environments have a facility for incorporating compiled and linked code from another programming language. Physical property calculation provides a good example, where existing, well-refined libraries could be incorporated into a simulator via the development environment.

C++ offers all of the facilities described above. It was developed in the early eighties as an object-oriented superset of the C language (Ellis and Stroustrup 1994). A class in C++ is a generalisation of the ANSI C structure type. It has been comprehensively tested for a variety of programming applications and an international standard, ANSI C++. Several powerful development environments are available, most of which contain facilities for file management, debugging and class/object browsing. Through its subset, ANSI C, several numerical software libraries are available. C++ is considered a hybrid language because object-orientation is optional. Employing a combined object-oriented/procedural approach can incorporate existing C software libraries. A "pure" object-oriented language (Eiffel, for example (Meyer 1992)) requires the programmer to consider every entity in a program as an object. As the object-oriented paradigm has become dominant in the software industry, C++ has taken over as the most commonly employed object-oriented programming language.

```

#include <math.h> //include header file for math functions

class Complex_Number{ //start class definition
    public:
        double re,im; //declare real and imaginary parts
        double mod(void); //declare member function to
                        //calculate modulus
        double arg(void); //declare member function to
                        //calculate argument
        //other member functions would be below for
        //addition, multiplication etc
        ...
        ...
        ...
}; //end class definition

double Complex_Number::mod(void){ //start definition of member
                                //function 'mod(void)'
    return sqrt( re*re + im*im );
} //end definition of member function 'mod(void)'

double Complex_Number::arg(void){ //start definition of member
                                //function 'arg(void)'
    return arctan( im/re );
} //end definition of member function 'arg(void)'

```

Many other member functions could be defined for the class, for example a function to print out the real and imaginary parts. C++ also permits the various operators (+,\*, =,< etc.) in C to be redefined. This is known as **operator overloading**. Overloaded operators for the **Complex\_Number** class would permit simple arithmetic such as  $z = x*y$  to be performed directly. An obvious application of classes and overloaded operators is the use of vectors and matrices for numerical computation. The **Complex\_Number** class can be used in the following way:

**Complex\_Number** class would permit simple arithmetic such as  $z = x*y$  to be performed directly. An obvious application of classes and overloaded operators is the use of vectors and matrices for numerical computation. The **Complex\_Number** class can be used in the following way:

```
#include <iostream.h> //include header file for standard C++
                        //input/output
void main(void){ //start C/C++ program

    Complex_Number x;

    x.re = 1.2; //set real part
    x.im = 2.0; //set imaginary part
    //cout below is the standard C++ output device

    cout<<" \n Mod x is " <<x.mod();
    cout<<" arg x is " <<x.arg()<<" radians" ;

} //end C/C++ program
```

Some features of C++ code should be emphasised. (A full description of C/C++ syntax is available in any of the common C/C++ textbooks, such as Ellis and Stroustrup (1994)). Attributes or members of objects are accessed with the '.' operator, for example, `x.re` and `x.mod()` above. The ';' operator denotes the end of a code statement. C++ provides three levels of access to attributes of an object. The broadest is `public` access or scope, where any member of an object can be accessed through the '.' operator. This is the same as the **ASCEND** language described in the previous chapter and is used at the start of the class declaration above. The next level is called `protected` access. Any attribute declared `protected` may only be accessed by objects of the same class or classes lower in that hierarchy. The most restricted access is `private`. Members declared `private` cannot be accessed with the '.' operator at all and are only available within the class or object structure.

## 2.2 Data Structure

Object-oriented programming is an information modelling process. It embraces three significant aspects of good software engineering, viz. *abstraction*, *modularity* and *information hiding*.

- *Abstraction*: Identifying and applying the broad structure and functionality of the components in terms of user-level concepts, details of which need not be specified until actually required.
- *Modularity*: Dividing the problem and solution into structured components with formal interfaces for communication between components.
- *Information Hiding*: Making the internal details of components accessible through the formal interfaces.

Historically chemical engineers have adopted a highly modular approach to process systems analysis. Physical plant modules and their associated unit operations can be well-represented by object-oriented principles as can their aggregated behaviour in the form of a process flowsheet. These concepts are not as conveniently implemented in traditional procedure-oriented languages.

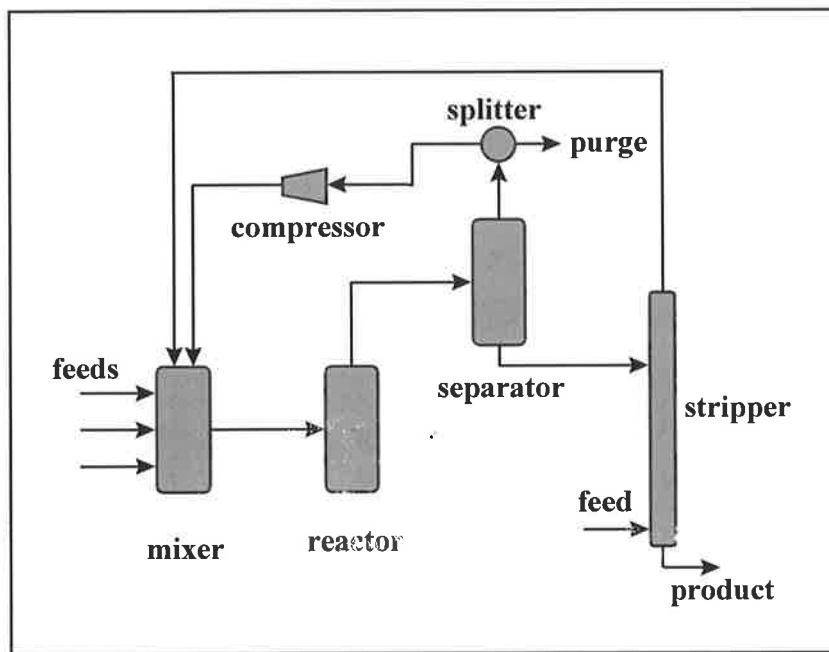
Simulation of a chemical process requires the modelling of two broad categories of information; physical and mathematical. Physical information is essentially the “things in the process”, such as process units, streams, chemical components, *etc.* Mathematical information includes the equations for the models, the model type (steady-state, dynamic, lumped-parameter, *etc.*) and the numerical analysis tools. It is not possible to develop the data structures for these two categories of information completely independently because at various points within a simulator they must interact. A simple example is the use of convergence blocks in a sequential-modular simulation. A convergence block becomes part of the process layout via stream variables, but it is really part of the solution tools and not the process structure. If the interaction between the process and mathematical data structures can be restricted to well-defined interfaces then a versatile, user-extendable simulation environment should be more easily realised.

The design of the data structures of this project is described below by applying the principles of *abstraction* and *modularity* firstly to the physical and mathematical information of a

process flowsheet and secondly to functional considerations. The application of the principle of *information hiding* is discussed in Chapter 3. The physical information of the process flowsheet structure is considered first followed by the mathematical information.

### 2.2.1 Physical Information

Consider the flowsheet drawn below in Figure 2.1, the Tennessee Eastman Challenge Problem (Downs and Vogel, 1993).

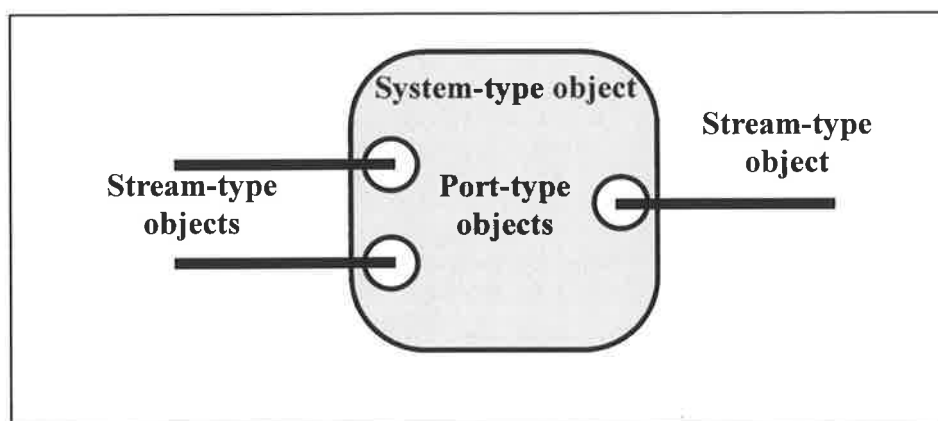


**Figure 2.1: Simple layout of Tennessee Eastman Process.**

Many of the basic parent classes required to represent the physical structures in typical chemical processes can be determined by examining the key attributes of this flowsheet. The flowsheet can be decomposed into units. These units could possibly be further decomposed (internally) into sub-units. For example the stripping column could be subdivided into a set of trays. The units receive inputs and produce outputs through streams. The flowsheet's primary function of synthesising a product from feeds is the result of specific functions of its constituent units. The various units mix, react and separate various chemical components. A detailed understanding of each unit's functionality is not apparent from the flowsheet (for example, reaction kinetics or order) but it is sufficient at this stage to identify a general functionality as a key property of a unit.

Four physical attributes common to most process flowsheets can be identified and used to define three basic parent classes that may contribute to object-oriented simulation. The first class is a group of processors that produce an output response from an input. In this work, the resulting C++ class is named **System**. The second class is for the connectors, named **Stream**. The third class is for the set of chemical components that a flowsheet manipulates. The relevant C++ class is named **Component**. The attribute of functionality will be discussed later in this chapter.

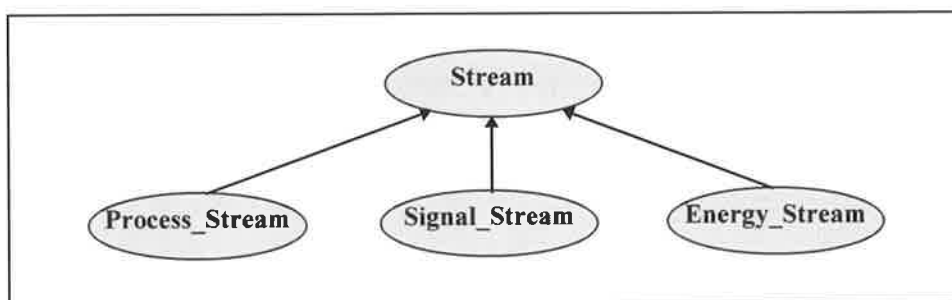
The **System** class is in principle similar to the **Generic-Unit** or **Model** classes of other work reviewed in Chapter 1. Given the varying levels of complexity contained in a flowsheet it is clear that *a System-based object must be able to contain other System-based objects*. A flowsheet contains unit operations and may be considered a single complex unit operation. Connections between **System**-based objects are made with **Stream**-based objects. However, this is not sufficient in an environment designed for user-extension of process models. A point of connection for each **Stream**-type object is required. While it is possible to manually code in the connections between the output variables of one unit and the input variables of another, it is simpler and more efficient to create classes of **System-to-Stream** interfaces that can perform the connection automatically. In a similar manner to other work reviewed in Chapter 1, a generic parent class named **Port** is introduced to provide the interface between **System** objects and their connecting **Streams**. *The set of Ports for a System object defines the boundaries of that System*. A **System**-based object would own a set of **Port**-based objects, with the connections made through **Stream**-based objects, as in Figure 2.2 below:



**Figure 2.2: Basic connection example.**

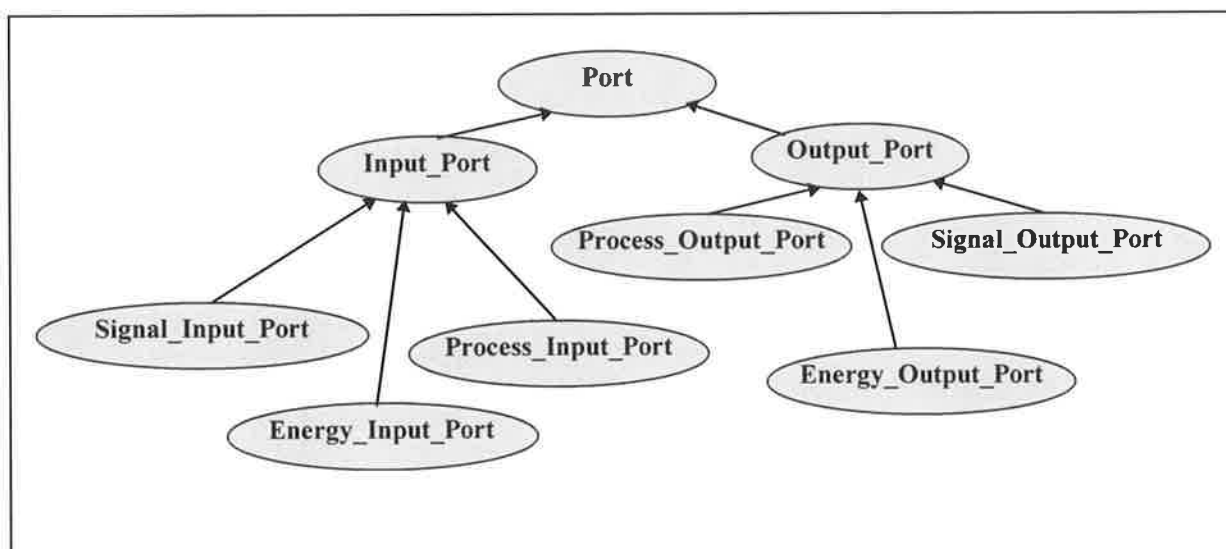


There are many different kinds of stream in a flowsheet. The most obvious are the process streams carrying material from unit to unit. Other kinds of streams include signals to and from controllers, or work streams where energy is transferred in and out of systems. The corresponding classes could logically be titled **Process\_Stream**, **Signal\_Stream** and **Energy\_Stream**. The **Stream** hierarchy defines a group of classes of simple connectors, illustrated in Figure 2.3.



**Figure 2.3: Stream class hierarchy.**

There is a one-to-two correspondence between **Stream** classes and **Port** classes. For each **Stream** type, an input and output **Port** class must be defined. While separate input and output **Port** classes are not absolutely necessary to model a connected system, in a chemical engineering context there is usually a direction associated with connections and the **Port** class is divided to reflect this. The inheritance tree for the **Port** class is illustrated in Figure 2.4.



**Figure 2.4: Port class hierarchy.**

Other advantages of a **Port** data structure will be discussed in section 2.2.3, Mathematical Information.

A complete object-oriented component and physical property system was considered outside the scope of the project given the time available. The other data structures have been made as simple and as flexible as possible to permit incorporation of more sophisticated data structures for components and physical properties. The **Component** class hierarchy provides part of the interface for physical property calculation. The design of the physical and chemical property class structures is presented at the end of this chapter in section 2.4, because it is more easily developed if the physical and functional aspects are considered together.

### 2.2.2 Simulator Executive

In order for object-based model classes to be created and used within the simulator by the modeller, a high-level simulator executive is required to process and manage the low-level model structures and behaviour. In most of the simulation projects discussed, the simulator executive is coded in a separate language from the model definitions, for example the OMSIM environment for the OMOLA language. The executive then translates the low-level model code into an equivalent high-level representation.

In this project, the executive is coded in the same language as the modelling classes. Therefore the equivalent high-level representation must exist in advance of any low-level structure. The high-level representation must be capable of connecting whatever low-level structure is created while maintaining its integrity and consistency. This is a stringent requirement. The equivalent high-level representation effectively becomes the executive itself.

While appearing complex, this multi-level structure is readily implemented with the class structure discussed. The examination of the basic classes in section 2.2.1 provides part of the solution. A **System** class was described that may contain other **System**-based objects. **System**-based objects must also connect to other **System**-based objects. This containment and connection may be implemented at the executive level and at the modelling level quite simply. At the executive level, the **System** class may contain a set of pointers to other **System**-type objects. A pointer in C or C++ is a type that contains the memory address of an

object (i.e. it points to where the object resides in memory). Through the memory address the object can be accessed. Object pointers are preferred because they require considerably less memory storage than extra sets of complete objects. A specific **System**-based object does not need to know in advance how many pointers there are or what type of **System**-descendant object they will attach to. In a similar fashion, the **System** class may also contain sets of pointers to **Input\_Port** and **Output\_Port** objects. Again, a **System**-based object does not require advance knowledge of the number or specific type of each connection. The low-level model descriptions may then assign the generic high-level data structure to their specific low-level object structures. The low-level structure is then available to the executive for analysis and manipulation. The high-level structure describes a **System**-based object potentially containing other **System**-based objects and sets of input and output connections. The actual connection objects are specifically described in the low-level structures. The code mechanisms for this are described in Chapter 3 and examples are presented in Chapter 4.

### 2.2.3 Mathematical Information

The mathematical information is part of the internal or invisible attributes of a process flowsheet. Traditional “black-box” simulators completely hide the mathematical information associated with a flowsheet and present the user with a set of predefined unit operation models. A simulator with a modelling facility must provide access to the mathematical information, preferably with a consistent set of building blocks for model development. In a similar fashion to the physical structure of a flowsheet, examination of the physical structure of a set of equations provides potential class structures for mathematical modelling. This is best illustrated with a simple example.

A vertical, cylindrical liquid mixing tank and valve are presented in Figure 2.5. Consider modelling the dynamic liquid holdups  $M_i$  of the tank with  $n$  components. It is assumed that the tank is open, is fed through stream  $F$  (with compositions  $z_i$ ), drains through the valve and the flow out  $L$  (with compositions  $x_i$ ) follows a square-root dependence on the pressure  $P$  at the bottom of the tank. The total mass in the tank is  $M_T$ .

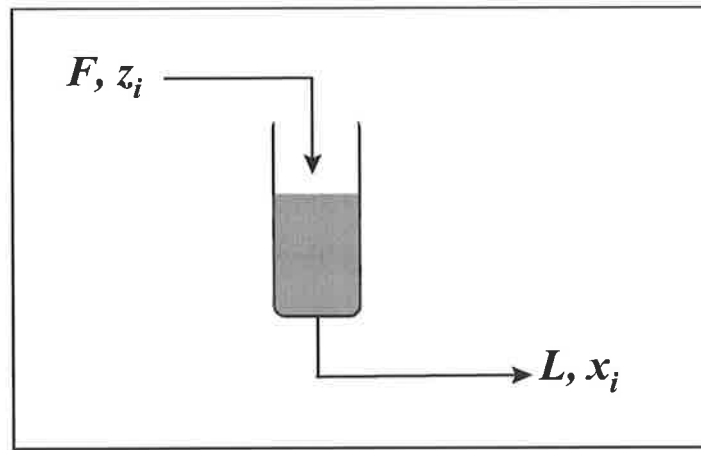


Figure 2.5: Simple draining tank.

The equations and variables for the dynamic mass balance are given below:

$$\frac{dM_i}{dt} = Fz_i - Lx_i \quad \mathbf{1\dots n} \quad (2.1)$$

$$M_i = x_i M_T \quad \mathbf{1\dots n} \quad (2.2)$$

$$\sum_{i=1}^n x_i = 1 \quad (2.3)$$

$$\frac{M_T}{A\rho} = h \quad (2.4)$$

$$P = \rho gh \quad (2.5)$$

$$F = C\sqrt{P - P_0} \quad (2.6)$$

**Variables:**  $x_i(n), z_i(n), M_i(n), M_T, L, F, h, P, P_0$

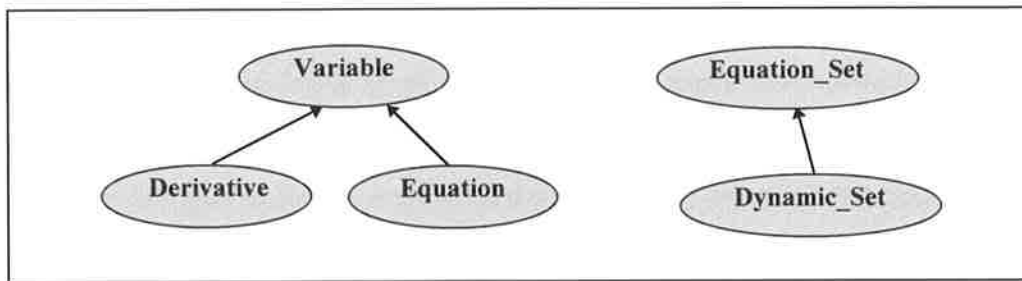
The attributes of the equations are in some ways similar to the attributes of a flowsheet. A flowsheet equation set consists of the equation sets of the units within the flowsheet. The equations constitute a set of mixed differential and algebraic equations (a DAE set). The individual equations are connected by variables, for example  $x_i$  appears in equations (2.1), (2.2) and (2.3). There are different kinds of variables in the equation set: state (dynamic), algebraic and derivative.

If the set above was to be used for steady-state and dynamic simulation, different equations and variables could take part in the solution. At steady-state, equations (2.1) and (2.3) could be solved, giving  $n+1$  equations with the time derivatives set to zero. The corresponding variables are  $x_i(n), z_i(n), F$  and  $L$ , of which  $n+1$  must be consistently specified for the set to

be solvable. If the steady-state height is specified, it is possible to solve for the component holdups and the valve constant also. This would be useful for specifying consistent initial conditions for dynamic simulation. The dynamic system requires all the equations, with the variables  $x_i(n)$ ,  $z_i(n)$ ,  $M_i(n)$ ,  $F$ ,  $L$  and  $M_T$ , of which  $n+1$  variables must be consistently specified for solution.

A class structure can be developed from considering the equation set's attributes. At the highest level we introduce a parent **Equation\_Set** class. This class represents sets of equations within **System**-based objects. If **System**-based objects can contain other **System**-based objects, logically **Equation\_Set** objects should be able to contain other **Equation\_Set** objects. The concept of **System**-based objects containing **System**-based objects combined with **Equation\_Set** objects containing **Equation\_Set** objects is a powerful tool for process simulation. The major advantages will be discussed in detail in section 2.3.1. The flowsheet equation set above would be constructed from two separate equation sets, one from the tank and the other from the valve. **Equation\_Set** is further refined into a **Dynamic\_Set** class for DAE sets. A **Dynamic\_Set** object can contain other **Equation\_Set** or **Dynamic\_Set** objects. The objective of steady-state and dynamic simulation requires **Dynamic\_Set** objects to be capable of steady-state and dynamic analysis.

A class **Variable** is defined for basic solution variables (state or algebraic). Two child classes are refined from this class, **Derivative** and **Equation**. **Derivative** is the class for representing derivatives in equations. It is not restricted to time derivatives, although this form represents the most common usage in chemical process simulation. **Equation** is the class for representing individual mathematical equations. It is capable of representing state or algebraic equations. The **Equation**, **Equation\_Set** and **Dynamic\_Set** classes have similar executive-level data structures to **System**. The **Equation** class owns a set of pointers to **Variable**-based objects that affect it. The **Equation\_Set** and **Dynamic\_Set** classes contain sets of pointers to the **Equation** objects that affect them, to create a pseudo-executive for analysing mathematical structures. The inheritance trees for the **Variable** and **Equation\_Set** hierarchies are drawn in Figure 2.6.



**Figure 2.6: Variable and Equation\_Set class hierarchies.**

In very strict object-oriented philosophy, an equation might not be considered a more refined version of a variable. It is important to realise that object-orientation is a tool as opposed to an end in itself. It is reasonable in some cases to deviate from a strict object-oriented structure if there are advantages to be gained. The advantages in the case of the **Variable** and **Equation** classes are primarily functional and relate to the analysis and solution of equation sets. These aspects are discussed in the next chapter.

The form of the equations must be considered. Are differential equations to be coded explicitly, such as  $\dot{y} = f(y, t)$  or implicitly as  $f(\dot{y}, y, t) = 0$ ? Are they likely to be combined if different models coded by different users are used together? The same applies to algebraic equations. A user might wish to code an explicit iterative equation in the form  $x = f(x)$ . Ideally, the class structure for variables and equations should have the capacity to cope with all of these in a mixed form.

As stated earlier, the mathematical and physical structures of a simulator must interact at clearly-defined points. Identifying the interactions helps to determine the interfaces and further refine the simulator class structure.

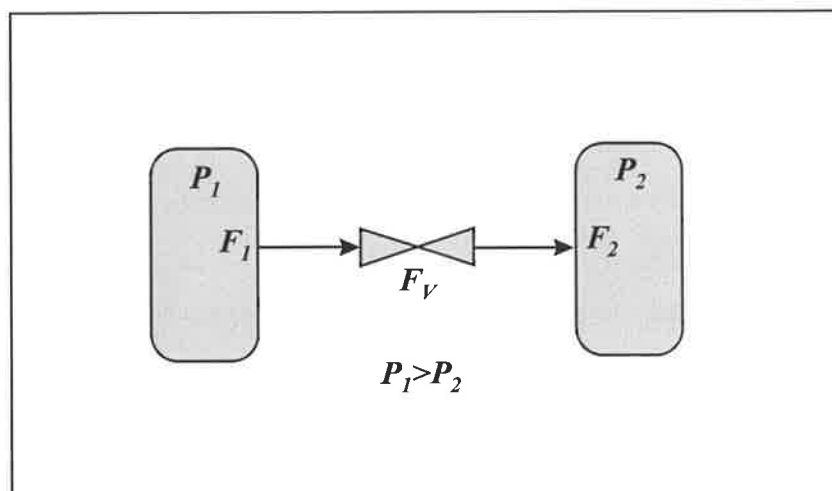
- **Unit Inputs and Outputs**

A unit will have sets of input and output variables, such as the  $(F, z_i)$ , and  $(L, x_i)$  sets of the tank example. Where do the variables reside? Are they an attribute of the connecting stream or are they part of the unit that they enter or leave? The variables in a stream are really defined by the unit that is the stream's source; even a feed stream must come from somewhere. Unit input and output variables in the current work are therefore considered to be an attribute of the process unit, not the stream. The stream-

variable approach probably originated from a desire to keep memory storage to a minimum in procedurally-programmed simulators. In the early stages of this project, a prototype stream-variable structure was developed and tested. However, the potential storage efficiency gains conflicted with the requirement of simple model development. In addition, a stream-variable structure blurs the boundary of the **System** class.

If a unit is to own its input and output variables, some sort of connection is required between the inputs of any unit and the outputs of its source. The logical connection mechanisms are the **Port** classes. In a process-unit sense, the variables or contents of a stream appear at the connecting flanges of the input pipes and disappear at the flanges of the output pipes. The input port of the tank model would access the  $(F, z_i)$  variables and the output ports would access the  $(L, x_i)$  variables. Other process stream variables, such as temperature and pressure could be included in the sets also. The connection mechanism between the variables, ports and streams can be provided by member functions of the respective **Port** and **Stream** classes.

Another aspect of unit inputs and outputs must be considered. The direction of mathematical information does not necessarily follow the assumed direction of connection between systems. An illustration of this point is a simple flow-restriction valve between two vessels, as drawn below in Figure 2.7.



**Figure 2.7: Flow restriction valve between tanks.**

A reasonable flow equation (ignoring density effects) is:

$$F_1 = F_2 = F_V = C\sqrt{P_1 - P_2} \quad (2.7)$$

where  $F_i$  denotes flow,  $C$  is the valve constant and  $P_1$  and  $P_2$  are the vessel pressures. The valve determines the flow in and out of the two vessels, but within the valve model the vessels define the upstream and downstream pressures. The material flows along the direction of the connections but the information flows in the opposite direction to the connection into the second vessel. The data structure must therefore permit bi-directional information flow. If the connections between input and output variables can be made directly with the variables themselves the data structure should become simpler. This could be achieved with **Port** and **Stream** class member functions. The **Stream** hierarchy then becomes a group of very simple connector classes with minimal structure. Ideally, the **Variable** class should contain the structure and functionality required to be *invisibly* either an input or an output variable.

The advantages of input-output connections between **Variable** objects are further demonstrated by examining stream compositions in the tank example above. The composition of the two streams will not change as the process material travels from the first tank, through the valve and into the second tank. In a simulation environment where general unit models are likely to be employed for a variety of purposes, the tank and valve models are likely to contain their own sets of composition **Variables**. Assuming that compositions are modelled as a composition vector, at least three vectors of **Variable** objects will be present in the tank example; one for each tank and one for the valve. The principle of flow direction suggests that the outlet of the first tank defines the composition. Therefore the second tank's composition **Variables** could connect to the valve's composition **Variables** which in turn could connect to the outlet composition of the first tank. This connection scheme is logical and provides two very important advantages. Firstly, it can provide consistent evaluation of the value of a particular **Variable** object. The **Variable** class may be easily coded to evaluate connections if required. The number of connections is irrelevant because several connections may simply evaluate further down a connected chain until the end



is reached, for example from the second tank, to the valve, to the first tank. This ensures consistent numerical evaluation and facilitates model coding, if the evaluation is invisible to the model developer.

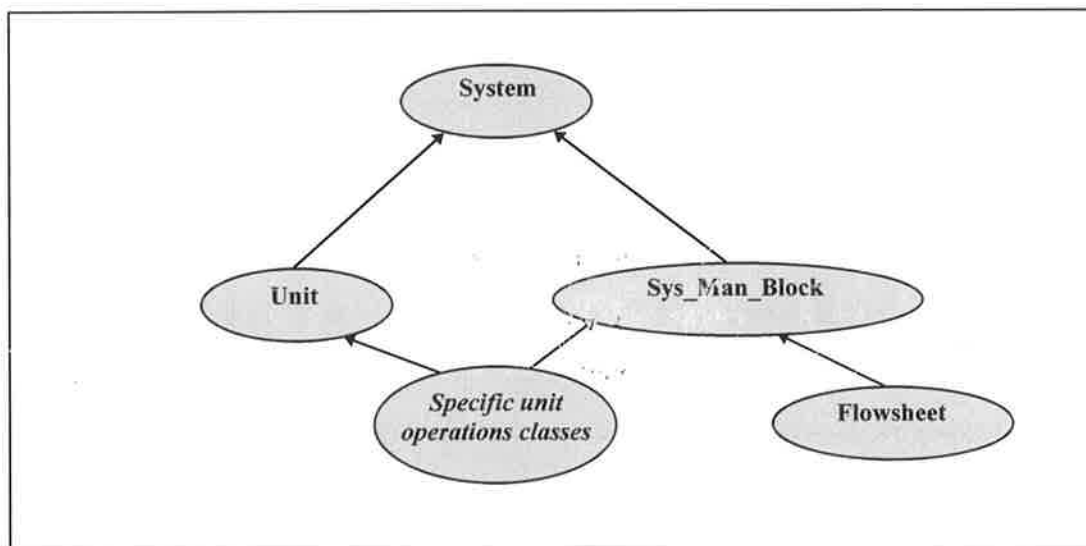
Secondly, connections promote simple and consistent analysis of **Equations** and **Variables** for problem specification. Consider the analysis of the tanks and the valve in an equation-oriented simulation. If the first tank is analysed, the outlet composition **Variables** will be analysed as part of the tank. If the valve is then analysed, the inlet composition **Variables** will have already been examined with the first tank. A connected **Variable** structure enables the simulator executive to immediately trace the connections back to the tank and note that the **Variables** have already been analysed and may be ignored. A similar procedure would occur in the analysis of the second tank.

- **Flowsheet and Complex Unit Equation Sets**

A flowsheet equation set or an equation set in a complex unit will be made up of many subsets of equations. The subsets will be owned by the subsystems of the flowsheet or units. A flowsheet has the ownership of the other **System** objects it contains and the equation set associated with a flowsheet therefore owns the equation sets of the other subsystems. This implies an interaction between the **System** and **Equation\_Set** classes at the executive level. This interaction must be consistent irrespective of the model type or structure in order for the simulator to provide a modelling facility. The interaction is most easily provided by an extension to the executive-level structure described in the previous section. The **System** class can contain pointers to **Equation\_Set** and **Dynamic\_Set** objects. Low-level models will contain a specific **Equation\_Set** and/or **Dynamic\_Set** object which the executive structure may be assigned to. The low-level mathematical structure is then available to the **System** executive via the mathematical pseudo-executive described earlier.

**System**-based objects containing other **System**-based objects forms a nested tree structure (for example, a flowsheet containing unit operations). By extension a **System**-based tree may contain other **System**-based trees. The structure of a **System**-based tree is sufficiently complex to justify a more refined class to specifically manage

the tree structure. The class is named **Sys\_Man\_Block**, an abbreviation for System Management Block. It inherits directly from **System**. The majority of the **System** class' functionality must be redefined to operate on the branches of the tree. The **Sys\_Man\_Block** class is designed for modelling with and managing trees of **System**-type objects in an arbitrary fashion, without necessarily adhering to flowsheet-type layouts such as input and output streams. **Sys\_Man\_Block** is refined into a **Flowsheet** class. A class for modelling with a single **System**-based object is also introduced, named **Unit**. The **Unit** class is designed for modelling basic unit operations and inherits from **System**. The class inheritance tree for the **System** hierarchy is drawn in Figure 2.8.



**Figure 2.8: System class hierarchy.**

- **Physical Structures**

An undefined area of interaction exists when the basic physical building block classes are not sufficient to describe what the user-developer wishes to model. An example is the characterisation of a biochemical process mixture. The nature of many biochemical systems may be so diverse that there are likely to be circumstances where some aspect of a biochemical mixture is not catered for within a simulator. In a system that provides user-extension, the user-developer should be able to modify existing physical modelling classes, create new ones or use existing class structures in a different way. The creation of a new type of mixture might require custom physical

property calculation and new **Port-Stream**-based classes. Within the data structure proposed this is relatively simple. Custom calculation methods can be incorporated by writing and compiling C functions or new C++ classes that operate on the existing structures. With the mixture and stream variables owned by the units, the user-developer only has to develop new **Port** and **Stream** classes that map input and output **Variable** objects to each other. It could also be possible to use existing **Port** and **Stream** classes for an unconventional purpose because **Variable** objects only interact mathematically within a **System** boundary. If a **Port-Stream** type is used in an unconventional way, it cannot be connected to conventional applications of the type, because the **Variable** connections in the conventional and unconventional applications will be different.

## 2.3 Functionality and Behaviour

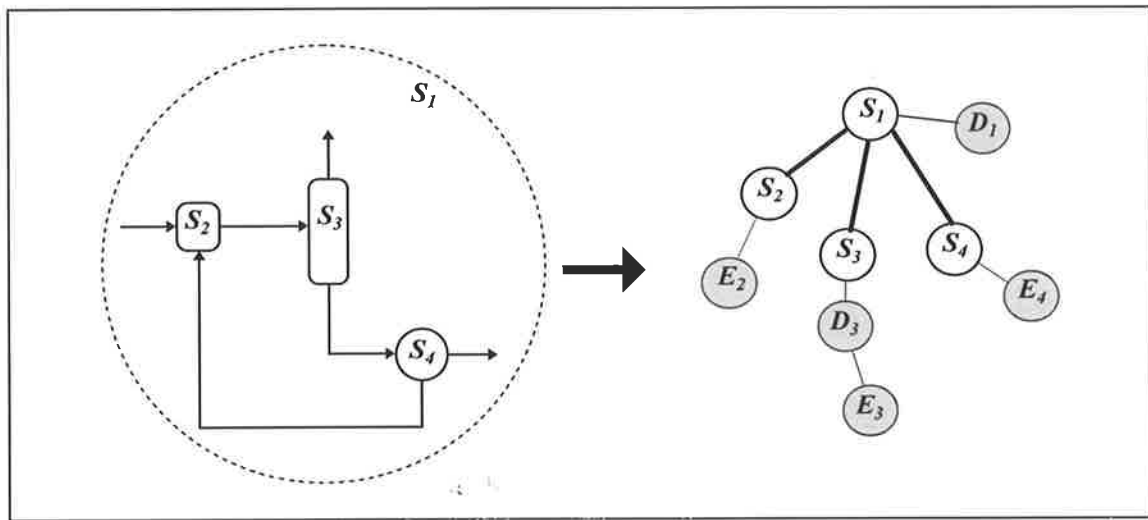
To this point the structural aspects of the simulator and the information in a flowsheet have been considered. Some functional aspects have been considered briefly. In this section, the major functional requirements of the physical and mathematical structures will be examined. In some cases, functional requirements or behavioural changes dictate additions or modifications to the data structures.

### 2.3.1 Structural Analysis

The use of **System**- and **Equation\_Set**-hierarchy objects to model a flowsheet of unit operations creates a multi-level tree of objects. **System**-based objects own **Equation\_Set** objects. **Equation\_Set** objects own **Equation** objects which in turn own **Variable**-based objects. This creates a connected tree from a top-level **Flowsheet** object down to individual **Variable** objects. Initially, only the physical structure tree will be fully connected as a result of the flowsheet connectivity. The mathematical structure tree will be composed of smaller trees within units or plant sections. In this project the mathematical tree is constructed after the flowsheet connectivity is defined and the unit parameters are set.

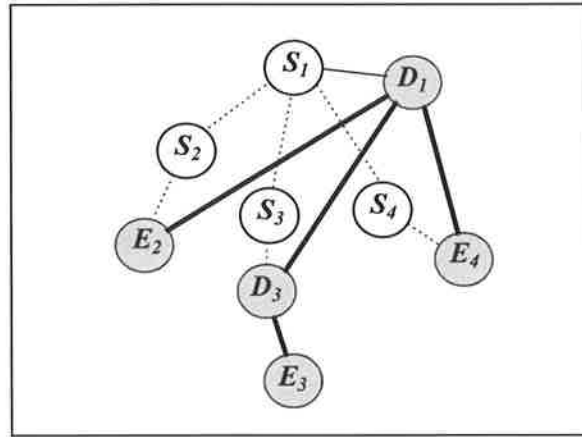
An example flowsheet and its tree are drawn below in Figure 2.9. The symbols  $S_i$  denote objects from the **System** hierarchy and  $D_i$  and  $E_i$  denote **Dynamic\_Set** and **Equation\_Set** objects respectively. Note that the  $S_i$  are not directly objects of the **System** class, they are objects of classes derived from **System**. The flowsheet is  $S_1$ , a mixer, a flash and a splitter are

units  $S_2$ ,  $S_3$  and  $S_4$  respectively. The heavy lines on the tree indicate the physical structure connections of the flowsheet.  $S_1$  contains  $S_2$ ,  $S_3$  and  $S_4$ . Each **System**-type object owns one or more **Equation\_Set**-based objects. The ownership is indicated by the light lines linking the  $S_i$  to the  $D_i$  and  $E_i$ .  $S_1$  (the flowsheet) owns a **Dynamic\_Set** object because some of its subsystems contain **Dynamic\_Set** objects.  $S_3$  has a composite **Dynamic\_Set** object ( $D_3$ ) with a sub-set of algebraic equations ( $E_3$ ).



**Figure 2.9: A flowsheet and its connected System-based tree.**

The mathematical tree is not yet connected. The first step towards a fully connected mathematical tree is to connect the input and output variables to each other through the **Port**-based objects. This can be achieved with a setup function in each **System**-type object. After this,  $D_1$  still does not know about the existence of the other sets of equations in the subsystems of  $S_1$ . The physical structure for  $S_1$  contains  $S_2$ ,  $S_3$  and  $S_4$ , so connection should start with the physical structure. The optimal way to connect the tree structure above is with a depth-first traversal. Both the physical and mathematical trees must be traversed. The end of a physical branch must be reached before a mathematical branch may be analysed and connected. The traversal order is  $S_1, S_2, E_2, S_3, D_3, E_3, S_4, E_4, D_1$ . Even if a node on the tree is visited, it will not be analysed until all nodes below it are visited and analysed. The *analysis* order is therefore  $E_2, S_2, E_3, D_3, S_3, E_4, S_4, D_1, S_1$ . This analysis automatically supports any level of model decomposition. The connected mathematical tree that results is illustrated below in Figure 2.10. The heavy lines now indicate the mathematical tree connections.



**Figure 2.10: Connected mathematical tree of flowsheet in Figure 2.9.**

Sets  $E_2$ ,  $D_3$  and  $E_4$  have effectively become “extra” sub-sets of  $D_1$ . A distinction is made between an “extra” sub-set and a plain sub-set. An “extra” sub-set is appended during a problem analysis whereas a plain sub-set is explicitly attached prior to the problem analysis as part of a unit model definition, for example  $E_3$  attached to  $D_3$ .

Once the mathematical tree is connected, it can be traversed on its own to collect the **Equations** and **Variables** for the flowsheet. The physical and mathematical structure could be traversed in a similar manner to the previous connection step, however, traversing only the mathematical tree requires less nodes to be visited. The **Equation\_Set** objects are collected in the order  $E_2, E_3, D_3, E_4, D_1$ .

In line with the objective of providing both steady-state and dynamic solution methods, the analysis and collection steps should create sets of **Equation**, **Variable** and **Derivative** objects or object pointers that can be manipulated by various solving routines. **Dynamic\_Set** objects should be capable of both steady-state and dynamic analysis and collection.

Depth-first analysis and collection ensures that any **Equation\_Set** object at any level in the tree automatically contains or owns the **Equation\_Set** objects below it on a physical and mathematical branch (a node-branch). For example, in order for set  $D_3$  to be collected into the set  $D_1$ , set  $D_3$  must have already collected set  $E_3$ . In addition, every **Equation\_Set** object contains its own set of **Variable** and **Equation** object pointers after analysis. Therefore these **Equation\_Sets** may be used to solve solitary units, as in a sequential-modular simulation. At

the same time, the **Flowsheet** object contains its own **Equation\_Set** object which has its own set of **Variable** and **Equation** object pointers. The **Flowsheet**'s **Equation\_Set** object may be used independently of the **Equation\_Set** objects in the process units to solve the whole system. The implementation of interchangeable simulation techniques is discussed further in section 2.3.6.

Individual units can be solved within a flowsheet or the plant may be divided into plant sections. Units could then be optimised individually (say, adjustment of a design specification) and then solved again within the overall flowsheet structure. The set of variables and equations for the parent flowsheet can remain static while individual node-branches are reanalysed and simulated separately. Small plant sections can be constructed out of units, simulated and used as building blocks of a larger flowsheet. A complete description of the depth-first algorithms with flow diagrams is presented in Chapter 3.

### 2.3.2 Equation Evaluation

Numerical expressions must be evaluated in a simulator. The structure and functionality present in the objects determine how object-oriented numerical expressions are evaluated. Three options are considered here. The first has been explored in other work (for example, Lau 1992). Equations can be represented and evaluated as binary trees of mathematical expressions. Parsing textual mathematical expressions with an interpreter can create the trees. This offers the potential advantage of symbolic manipulation of equations and automatic access to the mathematical structure of the problem. A simple interpreter and parser was developed early in the work but was found to be several orders of magnitude slower than normal floating-point arithmetic when evaluating equations. There was also a considerable storage overhead. Hence, the development was not carried further although a very sophisticated implementation might have proved more effective. A possibility would be an interpreter that translated textual expressions into C++ code instead of connected tree objects.

The second option is the definition of overloaded operators (described in Chapter 1, section 1.2.3) for the **Variable** types. The numerical, logical and assignment operators (+, -, /, \*, = *etc.*) could be overloaded to act on objects of the **Variable** hierarchy. The main disadvantage of this approach is the enormous size of the code required to cater for every possible interaction

between **Variable**-type objects, conventional numerical types and the library of numerical utility functions supplied in C++.

The third option retains an object-oriented structure such as the **Variable** hierarchy but evaluates numerical expressions in conventional floating-point arithmetic as standard C++ arithmetic statements. This offers the structural information of the mathematical tree (but without symbolic manipulation) with the convenience of floating-point arithmetic. There is obviously an associated storage overhead with the structural information. No redefinition of operators or utility functions is required, which removes a portion of software maintenance. This option was adopted for the project. Other advantages of floating-point evaluation are described in section 2.3.4.

### 2.3.3 Model Evaluation

Mixed equation forms (explicit/implicit) have been discussed previously. If mixed equations are supported, the evaluation of unit models must be “intelligent” so that equations are evaluated correctly. The evaluation could be at two levels, one that the user defines and an invisible higher-level evaluation to cater for the different equation types in the equation sets. The user-defined level is similar to the traditional unit model subroutine or procedure, where the model could be in explicit or implicit form and steady-state or dynamic. The higher-level evaluation could examine the structural information of the equations and act accordingly. As discussed in Chapter 1, explicit equations are easily transformed to implicit form in the following manner:

$$x = f(x, y) \quad \textit{explicit} \quad (2.8)$$

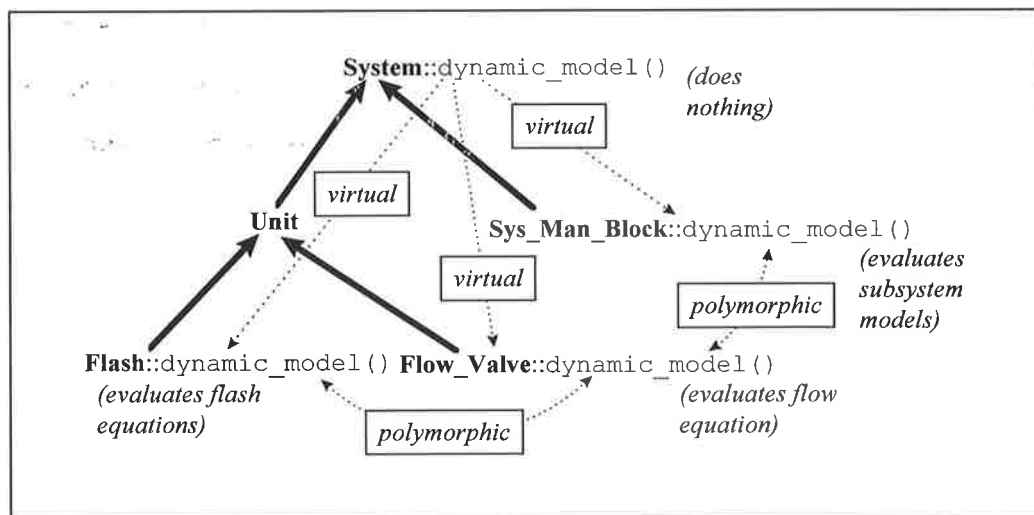
$$0 = x - f(x, y) \quad \textit{implicit} \quad (2.9)$$

This information is readily incorporated into the **Equation** class and is applicable to dynamic and steady-state equations.

The principles of object-oriented *polymorphism* and C++ *virtual functions* are well-suited to evaluation of model equations at the lower level. The **System** class can own a member function with a particular name, such as `dynamic_model()`, that does nothing at the **System** level. At the more refined level of a particular model, such as a flash, the

`dynamic_model()` function can be redefined to evaluate the equations of the flash. The model is evaluated by a virtual function that changes its behaviour as the inheritance tree becomes more refined. Different types of unit model then have polymorphic `dynamic_model()` functions across the **System** hierarchy, such as the different models for a flash, a mixer and a valve. These would be the low-level evaluations.

The C++ keyword *virtual* when applied to a member function means that the most refined implementation of the function can be run by calling the function at any level in the inheritance tree. For example, in a **Flash** class that is refined from **System**, by calling the function `dynamic_model()` at the **System** level, the `dynamic_model()` function of the **Flash** class is run. Hence, a **System**-type that contains other **System**-types (for example, a **Flowsheet** object) does not need to know in advance what the contained types are in order to run their models correctly. These concepts are illustrated with some example classes in Figure 2.11:



**Figure 2.11: Virtual and polymorphic model functions.**

The bold type and arrows indicate inheritance up through the **System** class hierarchy. The straight dashed arrows indicate the refinement of the virtual `dynamic_model()` function down the **System** hierarchy and the curved dashed arrows indicate the polymorphism of the `dynamic_model()` function.



The higher-level evaluations could be performed by the numerical methods. Different numerical methods require different evaluations, therefore the numerical methods should own the evaluation methods. This also separates the user's model definition from the model solution. The user should only need to specify the equation type, which is a trivial task.

#### 2.3.4 Behavioural Changes

This concept is generally restricted to dynamic simulation, where over the course of a simulation, unit operations might operate outside their "normal" mode. Such a change in behaviour may affect the progress of a simulation. A simple example is the saturation of a controller output. In the physical plant, the output may simply hold at the saturated value. In a process simulator, some mechanism is required that will enable the simulation to progress in spite of the saturation. The difficulty is numerical. The controller output will be a state or algebraic variable in the solution set. The sudden (discontinuous) freezing of a value could effectively remove a solution variable from the equation set.

The connected **Variable** structure described earlier may be modified slightly to provide an elegant solution. A process unit is a transformation mechanism. In traditional process control terminology it has a transfer function that produces a response to an input. The inputs to a unit are often outputs (or responses) from another but from the unit's frame of reference the inputs are only a forcing function. Consider a valve connected to a controller. The forcing function for the valve is the controller output. The controller output **Variable** object will be part of a **Flowsheet** object's **Dynamic\_Set**. The valve position input **Variable** object only connects to the controller output. This means that in the event of saturation it is possible to sever the connection between the valve input and controller output **Variable** objects without affecting the numerical solution. The controller output **Variable** object remains a solution **Variable**, and the valve position may be frozen at the saturated value. In the future, the controller output might return to a normal range, in which case the valve may be reconnected. Further functional requirements are apparent from the discussion. Generic virtual functionality must be provided that enables **Systems** to check for potential discontinuities or events and take appropriate action. The responsibility for the check must be on the **System** that owns the output **Variable**, because there might be many input **Variables** connected to one output **Variable**. The analysis of **Equation\_Set** objects must also trace connected **Variables** back to their source **Variable**, to ensure that if connections are broken, the correct

**Variable** is frozen and the solution set is not affected. The numerical methods are also affected by discontinuities. They contribute a virtual function named `disc_check()` which may be redefined in unit models to test for and flag discontinuities. The discontinuity checks and functionality are described in more detail in Chapter 3.

A more complicated example is a flash drum employed to roughly separate a process stream. During normal plant operation, the drum would contain a liquid and a vapour phase. However, if shutdown or upset conditions are simulated, unusual process conditions might create a single phase. The disconnection principle above is applicable to the output stream that “disappears”, but the model form might change also. Different model equations and variables may apply. Alternatively, careful modelling may yield a set of equations applicable to different phases in a flash calculation.

Some changes to equations may be accommodated relatively simply. If the numerical dimension of the system and the solution **Variables** and **Equations** do not change, a different evaluation form may be substituted without adversely affecting the simulation. If the numerical solution methods act solely on the values of sets of **Equation** and **Variable** objects, the form or method of evaluation of the value of an **Equation** is irrelevant. This is automatically catered for by designing **Equation** and **Variable** object evaluations to be based on floating-point arithmetic as described in section 2.3.2. Traditional `if...then` coding within the `disc_check()` function can assign different evaluations for the unit model. **Variables** may be effectively rendered constant by substituting a simple linear evaluation for an **Equation**, for example `e(1) = x() - 6.0` to hold the value of **Variable** `x` at 6.0 during the simulation.

Such a substitution is applicable to the controller saturation example above. If a different evaluation was substituted for the controller output signal equation at saturation, the `disc_check()` function is required to evaluate the “true” controller output to determine if the controller is to be reconnected.

### 2.3.5 Numerical Methods

The numerical methods available in a simulator should be as independent as possible of the process and mathematical structures to permit simple addition of new methods or modifications to the data structures. The requirement of operating on floating point values is dictated above. An obvious question is:- Should these values be conventional floating-point arrays, or accessed through arrays of **Variable**-hierarchy objects? Floating-point arrays permit the simplest interfaces to existing third-party numerical code, especially with precompiled libraries where the code cannot be modified. Most of this type of software requires at least one interface function to be defined that evaluates the equation system, derivative values, Jacobian *etc.* Arrays of **Variable**-hierarchy objects permit the use of the structure and functionality of the objects. Examples include partitioning of equations or reordering prior to solution. The information about the problem's mathematical structure is contained within the **Equation** objects. The input-output connectivity between **Variable**-type objects could be employed to connect a set of **Variable**-type objects in the numerical methods to the sets of pointers provided by the **Equation\_Set** analysis methods. This would increase the amount of memory required. Third-party numerical code could still be incorporated through interface functions. The set of object pointers provided by the **Equation\_Set** methods could be manipulated directly. Minimal extra storage is required and the structure and functionality of the objects is accessible. This approach was adopted for the project.

Broadly, there are two types of solution method employed in this project: AE and ODE/DAE solvers. Pure ODE systems are rare in flowsheet simulation and are considered a simpler subset of DAE solution. DAE solvers employ AE solvers as part of their algorithm. AE solvers in turn employ linear algebraic equation solvers. The solvers require methods of evaluating the Jacobian matrix. Related to pure numerical solution methods are methods that can partition or reassign equations.

The various classes could be employed as parents or as building blocks of new classes. Nonlinear equation solvers often require linear equation solvers. A linear equation solver class could be a parent class of a nonlinear solver. In this case inheritance is used to propagate functionality rather than structure down the inheritance tree. Alternatively, the linear solver may be an object within the nonlinear solver class. Either approach would be effective. A strict object-oriented philosophy dictates the use of objects instead of parent

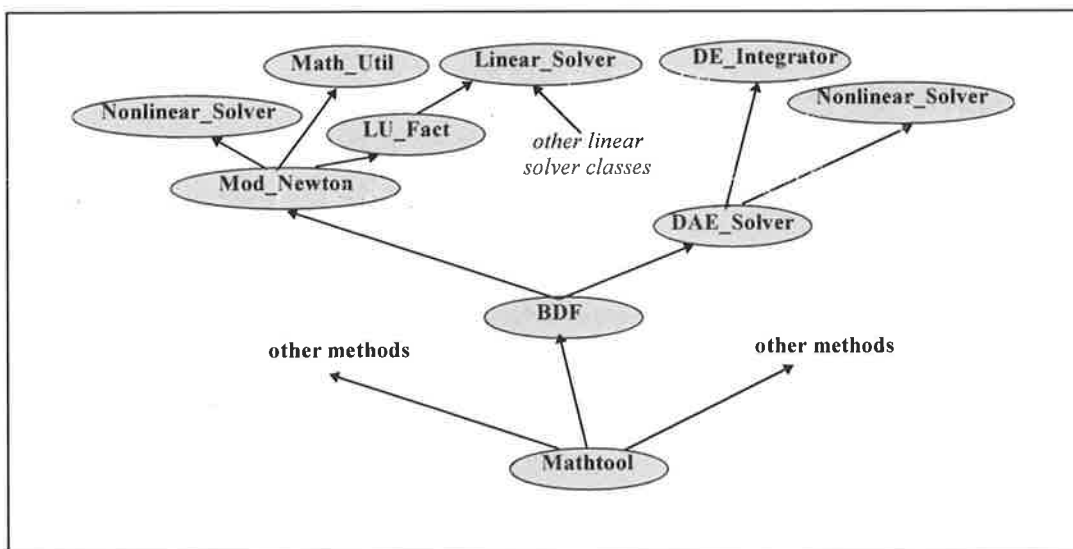
classes. However, object-orientation is a programming tool and not an end in itself. If a numerical method may be considered to be simply a repetitive procedure or algorithm, an inherited functional approach is valid. It is a difficult issue to resolve. For this project, the inherited functionality approach was adopted.

The interaction between the solvers and the flowsheet structure must also be determined. A numerical method object could act on a **System**-type object. This concept is similar to older Fortran-based simulators, where a flowsheet function, equations and variables are passed to a numerical method function for solution. Conversely a **System**-type object could own numerical tool objects that it employs for solution. This is more in line with the concept of a flowsheet owning methods to solve itself. Is the solution method then a separate entity with its own structure (i.e. an object) or is it only a service or method of the flowsheet? If the solution method is an object, it requires access to **System**-type functionality (the unit model functions) in order to solve the equations. This is easily achieved with a **System**-type pointer within the numerical method. If the solution methods are a parent of a class then the functionality is automatically available.

In this project, the numerical methods are implemented as a combined parent with the **System** class. The numerical method classes are designed to contain virtual function “mirrors” of some **System**-hierarchy model functions. The high-level structure permits independent development of the **System** hierarchy and the numerical methods. At the low-level, the virtual functions merge to provide a **System**-type model function for the numerical methods. The lower levels of the **System** hierarchy control the actual mathematical models because the models *must* be independent of the solution method. The numerical methods may contain whatever high-level functionality is required to drive the low-level model in order to solve the numerical problem.

To achieve independence, the numerical method classes and **System** class could become joint parents (multiple inheritance) of the **Unit** and **Sys\_Man\_Block** classes, because from these levels down specific functionality and hence solution requirements are identifiable. It can be argued that if the numerical methods are considered to have absolutely no structure, then object-orientation of them is not necessary and they could be implemented as pure C functions. However, on further examination there are some behavioural aspects that object-orientation can neatly organise.

The discussion above of various solver types implies a set of classes, for example linear equation solvers, algebraic equation solvers and differential-algebraic equation solvers. Multiple inheritance may be employed to produce, say, a class of nonlinear solver that inherits from a specific linear solver (e.g. LU factorisation) and a numerical utilities class. Alternatively, a single-inheritance structure, commencing with the numerical utilities and inheriting down into linear, then nonlinear and then differential-algebraic solvers is also possible. A numerical method class named **Mathtool** was implemented as a collector of all the parent numerical methods, analogous to a numerical methods library. A multiple-inheritance structure based on four parent classes was initially investigated. The four parent classes were **Linear\_Solver**, **Nonlinear\_Solver**, **DE\_Integrator** and **Math\_Util**. An example of a backward-difference differential-algebraic class (**BDF**) is provided below in Figure 2.12.

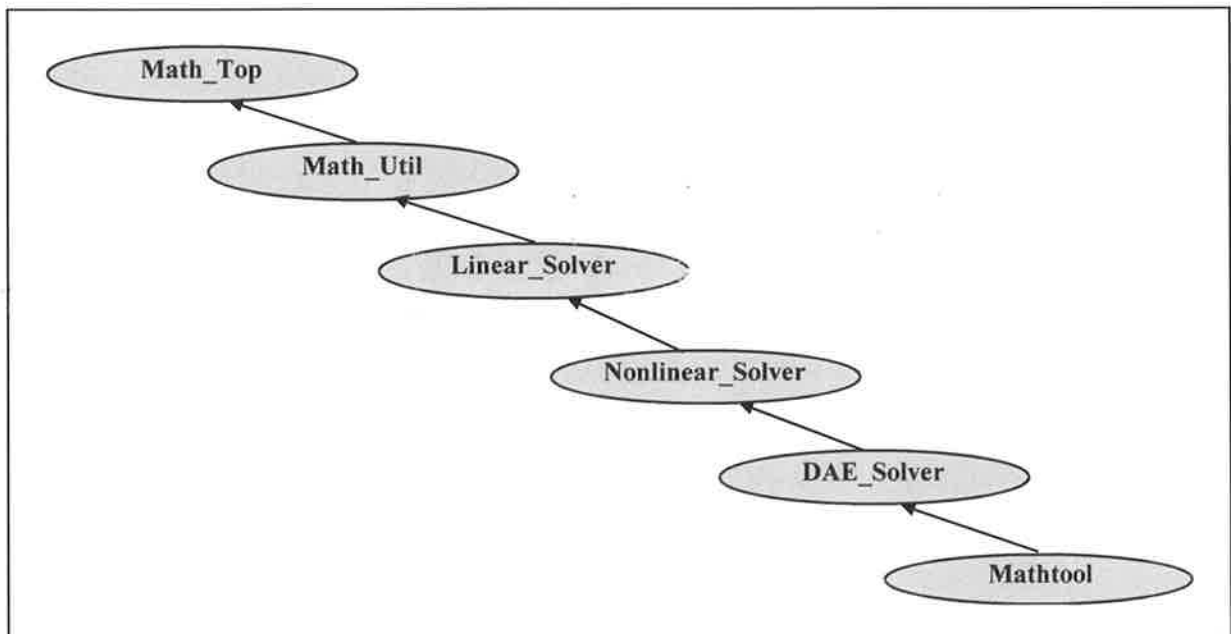


**Figure 2.12: Multiple inheritance numerical method class structure example.**

From Figure 2.12 it is clear that there will be several methods inheriting one or more of the four basic parent classes, such as **Linear\_Solver**. C++ provides a mechanism so that only one copy of a parent class actually exists in a complex multiple inheritance structure, if desired. This mechanism is called a *virtual base class* and is explained in detail in Ellis and Stroustrup (1994). The advantage of a virtual base class is that it resolves ambiguities with multiple parent classes and avoids duplication of class data. The four basic parent classes

become *virtual base classes* of the rest of the numerical structure. The class structure is reasonably complex for the **BDF** class. In spite of the *virtual base class* capability, the multiple-inheritance approach proved to be difficult to implement and manage and was abandoned.

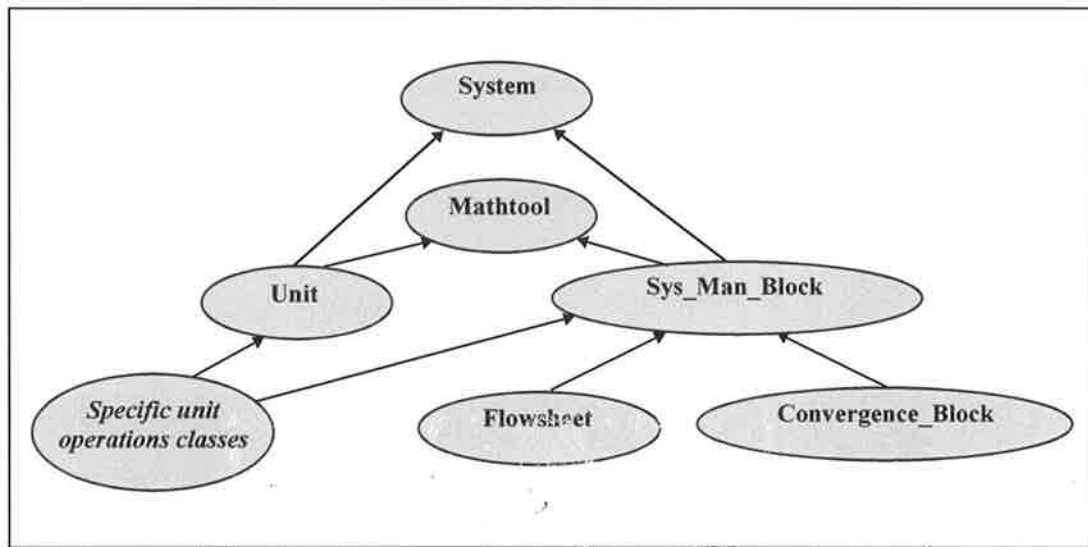
A single-inheritance approach was finally adopted for the numerical classes in the project. A top-level base-class was designed to contain basic structure and the virtual function “mirrors” discussed earlier. The class is called **Math\_Top**. **Math\_Util** inherits from **Math\_Top** and **Linear\_Solver** inherits from **Math\_Util**. **Nonlinear\_Solver** then inherits from **Linear\_Solver**. **Nonlinear\_Solver** is the parent of the **DAE\_Solver** class. **Mathtool** then inherits from the **DAE\_Solver** class. The class hierarchy is illustrated in Figure 2.13.



**Figure 2.13: Mathematical inheritance tree.**

The various methods for solving linear algebraic equations are in the **Linear\_Solver** class, all the methods for nonlinear algebraic equations are in the **Nonlinear\_Solver** class and so on. **Mathtool** then provides a separate interface class. The **Nonlinear\_Solver** class methods have access to a large variety of linear algebraic equation solvers from one parent class. The **DAE\_Solver** class has similar access to a variety of nonlinear equation methods, although generally a derivation of Newton’s method is applied in most cases. In spite of the diverse functionality presented at the **Mathtool** level, the extra structure is minimal and restricted to

the data in the **Math\_Top** class. The lower child classes provide functionality only. The functionality is more versatile than that provided by the multiple-inheritance approach and has a much simpler class structure. The combined **System/Mathtool** inheritance tree is drawn below in Figure 2.14. The **System** and **Mathtool** classes are multiple parents of the **Unit** and **Sys\_Man\_Block** classes. They each contain virtual functions with the same name, so that in lower-level classes there is a functional connection between the **System** and **Mathtool** parents. A new **Convergence\_Block** class, described in the next section, is also illustrated.



**Figure 2.14: Combined System/Mathtool class hierarchy.**

### 2.3.6 Interchangeable Simulation Techniques

One of the objectives of the project was to provide interchangeable steady-state simulation techniques. An **Equation\_Set** object can analyse itself and any **Equation\_Set** objects it contains. The interchangeable application of sequential-modular and equation-oriented simulation requires some additions to the simulator class structure. The dominant feature of sequential-modular solution is that it interferes with the flowsheet layout by tearing streams, although the interference is usually invisible to the user. Convergence blocks then manipulate the variables in the torn streams to converge the flowsheet.

If the functional aspects of a convergence block are examined, a very simple way of providing interchangeable solution methods is revealed. A convergence block requires a unit-by-unit iteration of the loop it is inside to perform its calculations: the unit model for a convergence

block is really a flowsheet or section of flowsheet. For the purposes of sequential-modular simulation, the flowsheet can be considered a subsystem of a convergence block. This suggests a more refined version of the **Sys\_Man\_Block** class, which is named **Convergence\_Block**. The class can own **Variable** objects, **Equation** objects and an **Equation\_Set** object. A **Flowsheet** object is a predefined **System**-type that can be set up and analysed for equation-oriented simulation. If the **Flowsheet** object is made a subsystem of a **Convergence\_Block** object, the methods to drive the unit model functions already exist. The **Convergence\_Block** object can then tear streams by reassigning the input and output variable connections of the relevant units. A convergence block in a simulator only acts on process stream variables. The simulator class structure provides a **Port** hierarchy that acts as the interface between **Stream**-types and **Variables**, so by interrogating the **Port**-type that the process stream attaches to, the input-output connections are accessible. Some sequential-modular simulators also attach convergence blocks to unit variables for design problems. This is only necessary if the unit models are explicit and the numerical methods based on iterative substitution. If explicit unit models can be converted to implicit models for equation-oriented solution then unit convergence blocks are not necessary.

A convergence block can employ the numerical methods for nonlinear equations available from the **Mathtool** class. Other methods can be provided in the **Convergence\_Block** class for finding tear sets and determining computation order of the **Flowsheet** object it owns (e.g. Roach 1996). An equation-oriented **Flowsheet** object is unaffected by changes in unit computation order. The **Convergence\_Block** solution methods could then apply either sequential calculations or equation-oriented solution, depending on convergence progress.

An interesting aspect of flowsheet solution should be considered here. It is clear that a **Convergence\_Block** could be employed purely for equation-oriented simulation by “switching off” the sequential-modular capabilities. The concept of a “super” convergence block is suggested, capable of driving all types of flowsheet simulation, including dynamic. The **Flowsheet** object within the **Convergence\_Block** could set itself up for dynamic simulation. The **Convergence\_Block** could then contain functionality to drive the **Flowsheet**’s integration methods. An alternative structure would be to place all of the **Convergence\_Block** functionality and structure into the **Flowsheet** class, in which case a



**Flowsheet** then has the capacity to initialise an equation-oriented simulation with sequential-modular iterations, converge the steady-state and then switch to a dynamic simulation.

Dynamic-modular analysis and solution could be performed in a similar fashion and implemented inside the **Convergence\_Block** or **Flowsheet** classes. This would provide a large variety of potential solution methods for steady-state and dynamic simulation. Implementation of dynamic-modular simulation is not explored in this project.

The actual **Convergence\_Block** implementation is less sophisticated. Sequential-modular and parallel-modular simulation are provided through the **Equation\_Set** object of the **Convergence\_Block** class. Equation-oriented simulation is provided by the **Flowsheet** object that the **Convergence\_Block** class owns. The user must supply the appropriate tear streams. The **Convergence\_Block** class is explained in more detail in Chapter 3.

#### **2.4 Chemical Components and Property Calculation**

The class for representing chemical components in the simulator is named **Component**. Chemical components usually occur as part of a mixture. Different types of mixture occur in a flowsheet, implying some sort of basic mixture class. In this project the class is named **General\_Component\_Mixture**.

The data structure and functionality for **Component** and **General\_Component\_Mixture** is based on attributes and methods for property calculation reviewed in Reid (1988). The discussion here covers conventional chemical components and is extended to unconventional (e.g. biochemical) components. Only systems where equilibrium between phases can be assumed are considered.

A conventional chemical component has several attributes. The most important is the component's name, and/or the molecular formula, through which the simulator's physical property service accesses the basic component data. Basic (invariant) pure component data includes the molecular weight, critical temperature, pressure and volume, boiling point, freezing point and enthalpies of formation at a standard reference condition, acentric factor, dipole moment *etc.* Other attributes possibly dependent on system or mixture conditions include ideal liquid and vapour heat capacity, enthalpy at system temperature and pressure,

enthalpy of vaporisation and liquid and vapour density. All of these attributes are required in a **Component** class for general physical property calculation. Methods for assigning values to the attributes are required, with the values extracted from a database or file. A user-defined component class would be useful for components not in the simulator database.

A component mixture is simple to represent physically as a **General\_Component\_Mixture** class: it contains a set of **Component** objects and a measure of their relative amounts (mass or mole fractions). Functionality is more complex. Which properties are likely to be required for process simulation? A very obvious one is mixture molecular weight. Likewise, thermodynamic properties are required. Mixture specific heat, enthalpy, entropy, Helmholtz and Gibbs energy and fugacity are examples. Vapour-liquid equilibrium calculations are also required in multi-phase mixtures, for example the  $K_i$  values in a flash calculation. Vapour-liquid equilibrium is related to the thermodynamics of the mixture through the fugacities. A basic parent class named **Properties** is introduced to provide the link to the set of relevant **Component** objects in the mixture. Separate classes named **Thermo** and **VLE** are defined for thermodynamics and vapour-liquid equilibrium because for simple approximations, the two calculations can be separated. The simple approximations are calculation of mixture properties from pure component properties and the assumption of Raoult's Law for vapour-liquid equilibrium where Antoine constants can be applied. The **Thermo** and **VLE** classes contain the virtual function declarations for the various thermodynamic and equilibrium calculations and inherit from the **Properties** class. Two child classes, **Simple\_Thermo** and **Simple\_VLE** are introduced for the simple approximations.

More sophisticated methods can be incorporated. Cubic equations of state (cubic EOS), which combine thermodynamics and vapour-liquid equilibrium can inherit from both the **Thermo** and **VLE** classes. The basic form of many cubic EOS is given by :

$$P = \frac{RT}{V - b} - \frac{a}{V^2 + ubV + wb^2} \quad (2.10)$$

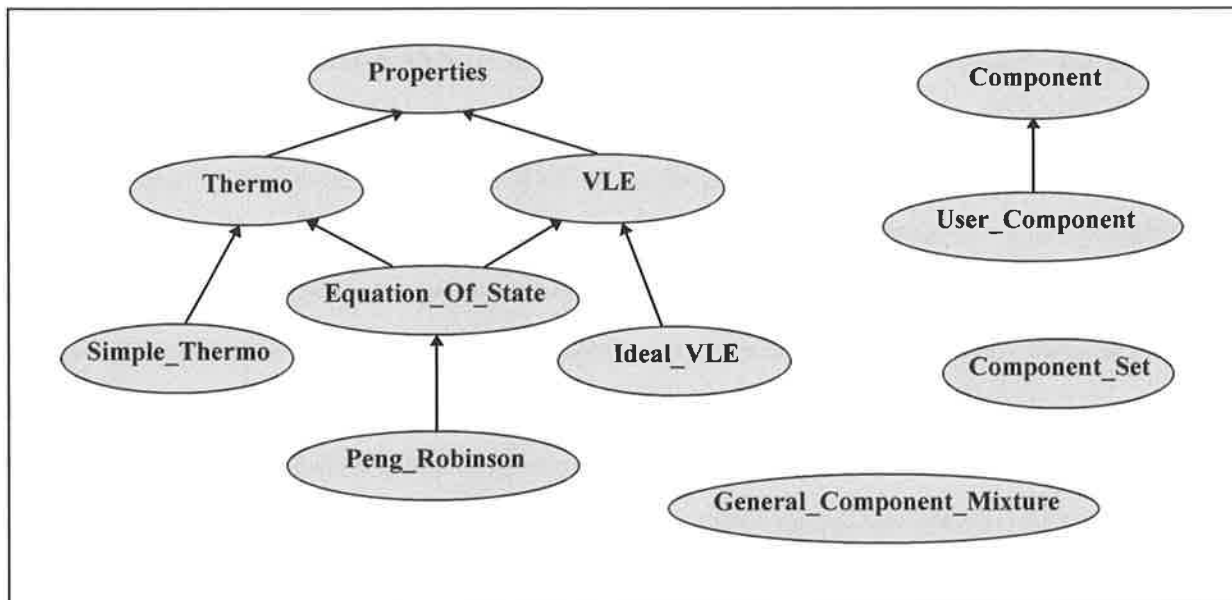
The parameters  $a, b, u$  and  $w$  vary depending on which particular equation is employed, e.g. van der Waals, Peng-Robinson *etc.* A further class structure is suggested here: a basic **Equation\_Of\_State** class containing the form of equation (2.10) with, say, a specific

**Peng\_Robinson** class (and others) that initialise  $a, b, u$  and  $w$ , appropriately. The enthalpy calculations and other properties discussed above would be member functions of the **Equation\_Of\_State** class.

In the early stages of the project, the modelling of separate phases was investigated. The **General\_Component\_Mixture** class could be extended to separate classes for different phases, **Liquid\_Mixture** and **Vapour\_Mixture**. These two classes could then be employed to define a **Vapour\_Liquid\_Mixture** class. Implementation of this class would be similar to the classes for numerical methods. The class could either inherit from both the **Liquid\_Mixture** and **Vapour\_Mixture** classes, or contain **Liquid\_Mixture** and **Vapour\_Mixture** objects. Objects would be preferable because a phase is physically identifiable.

The **General\_Component\_Mixture** class would contain a pointer to a **Thermo** object. This would enable different mixtures in different process units to access different thermodynamic methods if desired. The **Liquid\_Mixture** and **Vapour\_Mixture** classes would then have specific methods for liquid and vapour properties. The **Vapour\_Liquid\_Mixture** class would contain a pointer to a **VLE** object. The **Thermo** and **VLE** class hierarchies should have polymorphic functions for calculation methods so that the user-developer is presented with the same functional interface for the same property regardless of how it is calculated. For a cubic EOS, the **Thermo** and **VLE** pointers would access the same **Equation\_of\_State**-type object.

While this multi-level mixture class structure is logical, sophisticated phase modelling is unnecessary at this stage of development. A **General\_Component\_Mixture** class may provide similar phase calculation services with liquid and vapour calculation methods. The main physical property emphasis was therefore placed on the **General\_Component\_Mixture** class, without major development of the separate phase classes. The **General\_Component\_Mixture** class also contains pointer to a **VLE** object. The incorporation of a **Vapour\_Liquid\_Mixture** class into a physical property structure raises the question of whether a flash calculation is a member function of the class or a **System**-based unit operation model. In this project a flash is a **System**-based model. The physical property class hierarchies are illustrated in Figure 2.15.



**Figure 2.15: Physical property class hierarchies.**

Unconventional components, such as biochemical components retain some of the attributes above and add or delete others. Components common to “normal” chemical processing and bioprocessing could require completely different characterisations for each process type, for example ethanol. Attributes of bacterial cells, substrates and cell products or metabolites can be specified from an examination of the kinetics and thermodynamics of biochemical reactions (Roels 1983). The attributes include cell and product specific yields, specific heat dissipation, and degrees of reduction based on electron transfer. However, these are not physical properties, they are stoichiometric coefficients. Molecular weights and formulae are a common attribute, already catered for with the **Component** class. The meaning of the molecular formula of a bacterial cell type is different from a conventional component: you cannot isolate a molecule of cell, but you can isolate a whole cell. A macro-level approach was discussed in Chapter 1. It is similar to considering the cells as a semi-inert solid phase. For bacteria, cell strength, size and bulk density are useful properties. Cell size and strength are useful for homogenisation and centrifugation, and bulk density could be a useful parameter for calculating broth level in a fermentor.

A class for cell types could inherit from the basic **Component** class. Many of the basic attributes of the **Component** class are reusable, for example the density, specific heat and molecular weight. The **General\_Component\_Mixture** class could be a parent of a

biochemical mixture class. The mole or mass fractions of the class could become the fractions in a size or strength range for a cell type. Some of the thermodynamic methods would still be applicable. Vapour-liquid equilibrium calculations could also have applications to a biochemical mixture. A vacuum flash operation might be applied to separate volatile organics from an aqueous cell suspension, containing a **General\_Component\_Mixture** object and an inert biochemical mixture. A full biochemical class structure on this basis is not designed for the simulator. The development of physical and chemical property structures was restricted in order to keep the project to a manageable size and permit the inclusion of more comprehensive property facilities at a later date.

## 2.5 Summary

The design of the data structures for the simulator has been discussed in terms of the physical and functional characteristics of a process flowsheet and the objectives outlined in Chapter 1. These characteristics have been used to define class structures for three physical modelling areas and numerical methods which are reviewed below:

- *Structural.* Based on a **System** class hierarchy for representing unit operations and flowsheets. **System**-type objects are connected to other **System**-type objects with objects of a **Stream** class. The boundaries and interface between **System**-type objects and **Stream**-type objects is provided by a **Port** class hierarchy. A **System**-type can contain other **System**-type objects to create a tree. A class, named **Sys\_Man\_Block** is described for managing **System**-trees.
- *Mathematical.* Based on a **Variable** class hierarchy for representing equations, variables and derivatives, and an **Equation\_Set** class hierarchy for representing algebraic and differential equation sets. **Equation\_Set**-type objects can contain other **Equation\_Set**-type objects, similarly to **System**.
- *Chemical Components, Mixtures and Properties Calculation.* Based on three class hierarchies: **Component**, **General\_Component\_Mixture** and **Properties**. **Component** is the parent class for chemical components. **General\_Component\_Mixture** is a class for representing mixtures and **Properties** is the parent class for various chemical and physical property calculation types.
- *Numerical Methods.* A multiple-inheritance approach to the numerical methods proved to be unwieldy and a single-inheritance design was implemented. The

numerical methods are supplied as part of a collective **Mathtool** class that inherits from a variety of different solver classes.

The simulator executive has been designed to be a generic data structure that may be mapped to the low-level models. The main functional aspects of the three physical areas have also been considered. The functional aspects discussed were the analysis of physical/mathematical structures, evaluation of equations, evaluation of models, behavioural changes, numerical methods, and interchangeable solution techniques and property calculation. The C++ implementation of the class structures is discussed in the next chapter.

# CHAPTER 3

## C++ Implementation

This chapter describes the implementation (in C++) of the class hierarchies designed in Chapter 2. Structure and functionality are described concurrently. Complete class definitions are not presented in this chapter. Emphasis is placed on the attributes and functionality required for a user-developer to model unit operations and construct flowsheets. Examples of the application of the classes to modelling will be provided in Chapter 4. Class names are printed in bold type.

### 3.1 C++ Constructors and Destructors

Object-oriented programming is based on the creation of user-defined types. A C++ class definition describes a data structure and tells the compiler what is in an object of a particular class. In order for a user and the language compiler to create an object some further information is required. The user and compiler need to know the states of the data are when a new object is to be created (i.e. how to put it together). This information is contained within member functions named constructors in C++. The number of different constructor functions depends on how many different ways the class developer decides an object can be created. A constructor function has the same name as the class that owns it. Different constructors take different function arguments. If a constructor function is not defined for a class, the compiler attempts to define one itself. It is poor programming practice to fail to define at least a default constructor for a class. Complex classes nearly always require constructors to be defined.

Similarly, the compiler needs to know how to dismantle an object when it is no longer required. The function that dismantles the object and frees the associated raw memory is called a destructor. Only one destructor function may be defined for a class. A destructor function has the same name as the class that owns it but is preceded by a tilde (~). A destructor function may be non-trivial. The use of pointers to manipulate data structures and objects is common in C++. There are potential execution problems if an object is destroyed and other objects or data still contain references to the destroyed object. The data structure of

this simulator is designed so that interconnected objects are not destroyed until a simulation is complete and the entire program exits to the operating system. This significantly simplifies destructor coding, although a more sophisticated implementation would be required for more advanced development. The role of constructors and destructors will become clearer in the next chapter.

### 3.2 Vectors and Matrices

In practically all areas of programming, continuous collections and sets (arrays) of data types are required. Matrix and vector computation is standard in numerical work. A criticism of traditional implementations of arrays in computing languages is that a basic array in memory does not know anything about its size, starting index or finishing index. The application of object-orientation to address these deficiencies is obvious. A **Vector** class has been created for this project that contains all the necessary information about an array of values: the starting index, finishing index, memory location and functionality to inform the user when an attempt is made to access memory out of the array bounds.

C++ offers a further enhancement, which is the concept of a *template* class. A template class is basically a class for classes. Template classes are explained in Ellis and Stroustrup (1994). A template permits the **Vector** class to construct a vector of any type of object, so that a **Vector** class is defined for every other C++ class. Vectors of **System**-types, **Variable**-types, **integer**-types *etc.* can be created with a single, consistent declaration. Another advantage of C++ is dynamic memory allocation. This means that a **Vector**-type object can be declared without necessarily defining its size. The memory for the array of objects can be allocated in the future. In ANSI C and C++ this is called dynamic memory allocation. C++ has built-in error handlers in case sufficient memory is not available. The **Vector** class is a template class. A similar **Matrix** template class has also been defined.

Another feature of dynamic memory allocation is that it enables arrays to be lengthened or shortened. Only lengthening is implemented in this project. One application is the analysis of the **System/Equation\_Set** structure of a **Flowsheet** object where extra sets of equations are added to the **Flowsheet** object's set during the depth-first analysis. The lengthening functionality incurs no penalty for access time in the lengthened **Vector**. Another feature incorporated into the **Vector** class is the capacity for one **Vector** object to access another



**Vector** object's array, provided they are of the same type. This is useful for passing arrays of **Variable** and **Equation** object pointers to numerical methods. Objects from the **Vector** class will be employed in unit operation examples later in the thesis. Examples of the use of **Vector** and **Matrix** objects serve as a useful demonstration of some important aspects of C++.

A **Vector** object, called **x**, containing ten double precision elements, starting at index one and finishing at index ten is declared by:

```
Vector<double> x(1,10);
```

Elements are accessed by an integer argument in round parentheses, thus:

```
x(3) = 4.0;
```

Simple arrays in C or C++ are normally accessed with square brackets (e.g. `x[3]`). The array operator `[]` could be overloaded in the **Vector** template class to mimic the simple array operator, however the `()` operator is overloaded instead to indicate to users that they are working with a **Vector**-type object and not a simple array. **Vector** objects do not have to start at index one, they can start at any index greater than or equal to zero. The end index must be greater than or equal to the start index. A similar **Vector** object called **f**, containing ten **Flowsheet** objects is created by the code statement:

```
Vector<Flowsheet> f(1,10);
```

An unallocated **Vector** object **x**, containing double precision elements, is created by the code statement:

```
Vector<double> x;
```

The **Vector** object's size can be allocated later in the code by the statement:

```
x.build(1,10);
```

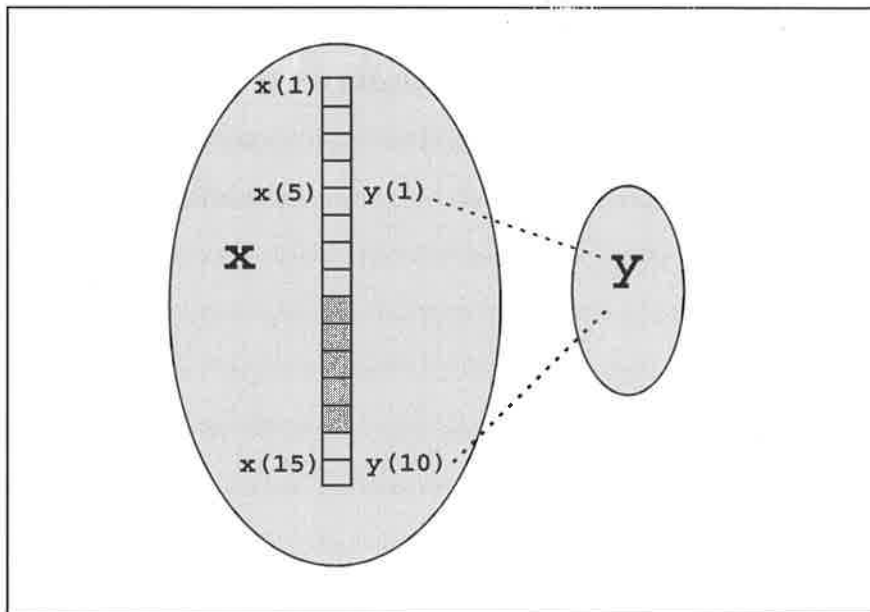
The `x` object above may be increased in size by the statement:

```
x.grow(1,5);
```

This adds five elements to the end of `x`, giving it an index range of `x(1)` to `x(15)`. Accessing another `Vector` object's array is demonstrated by the code:

```
Vector<double> x(1,15), y;  
x.sub_access(y,5,15);
```

Two `Vector` objects `x` and `y` are declared. Only `x` is allocated storage. The `Vector` `x` then assigns the unallocated array pointer in `Vector` `y` to elements `x(5)` to `x(15)`. An important point is that the elements of `y` default to `y(1)` to `y(10)` and not `y(5)` to `y(15)`. The principle is illustrated in Figure 3.1. The `Vector` `x` owns the allocated memory and `y` has access to part of `x`'s memory.



**Figure 3.1: Multiple access of Vector objects.**

This also illustrates the use of different constructors in a C++ class. Any type may be allocated into a `Vector` object, provided the type has a default constructor. A default constructor takes no arguments and generally puts an object together in the simplest way

possible. Inside the **Vector** class there is a constructor taking two integer arguments that define the start and end indices of the array. The constructor then allocates the required memory for the number of elements of the type passed to the **Vector**. This constructor runs when the declaration `Vector<double> x(1,15)` is made. The **Vector**'s default constructor (with no arguments) is run when the object `y` is created. This default constructor sets the start and end indices to zero and ensures that the array inside the object is null. The constructors and their arguments dictate how an object may be declared. It is not possible to declare a **Vector** object `x` with the statement `Vector<double> x(1,10,4)`; because no constructor function exists that takes three arguments. The corresponding destructor function for the **Vector** class deallocates the array's memory.

The **Matrix** class is similar. A **Matrix** object called `x`, with three rows and five columns of double precision elements is declared by the code statement:

```
Matrix<double> x(1,3,1,5);
```

The element in row three, column four is accessed and assigned by the code:

```
x(3,4) = 5.234;
```

The indices are placed within a single pair of parentheses to decrease execution time for the access operation. To overload the `()` operator to enable code such as `x(3)(5)` requires a matrix to be stored as a **Vector** of **Vectors**, which greatly increases the access time for an element by several factors. Slow access time is unacceptable, particularly for numerical computation.

The arrays in both **Matrix** and **Vector** objects can be erased with a member function named `clear()`. The object can be reallocated with the `build(i,j)` function described earlier.

Unit operation classes are created in the next chapter, which further demonstrates the complexity and importance of constructors.

### 3.3 Process Class Structure

This section describes the implementation of the **System**, **Stream** and **Port** class hierarchies in C++. Detailed descriptions of each class' functionality are provided in Appendix A.

#### 3.3.1 System Class and Descendants

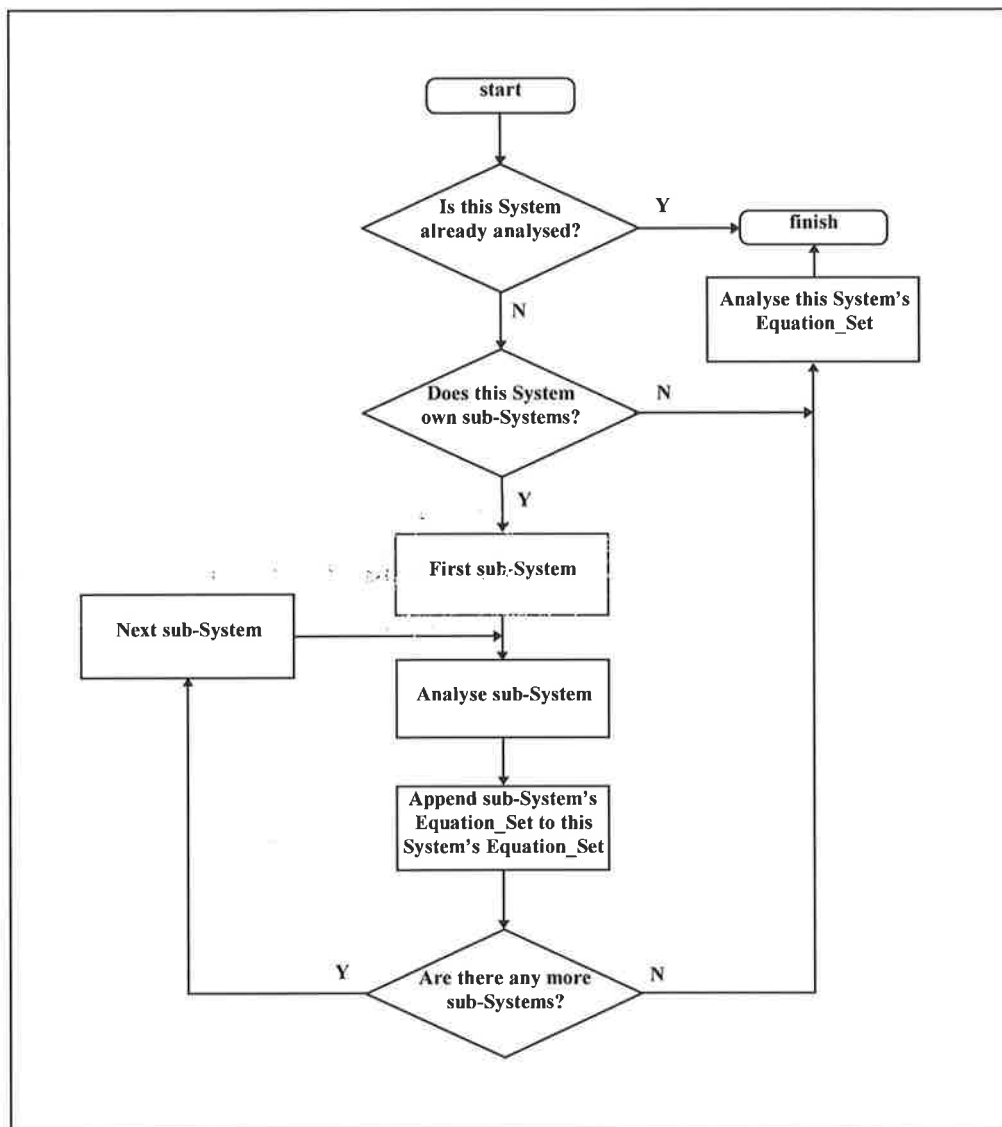
The **System**-based data structure is implemented at an executive and a modelling level as outlined in Chapter 2. The executive structure is the basic framework for process representation and may not be modified. This structure relies on the basic attributes of a connected physical system being similar. The majority of the executive structure is contained inside the **System** class definition. The low-level structure is user-defined and deals predominantly with object-based modelling. The low-level structure is implemented in classes derived from **System** and is demonstrated in the next chapter with examples.

At the high level, a **System** owns inputs, outputs, other **Systems**, **Streams**, steady-state and dynamic equation sets and a transformation model. These attributes are generic at this level. The physical attributes are implemented as **Vector** objects containing pointers to objects of the basic **System**, **Input\_Port** and **Output\_Port** classes. The mathematical attributes are pointers to objects of the **Equation\_Set** hierarchy.

The specific attributes must be invisible to the user and inaccessible except through a limited set of interface functions. This is an application of the software engineering principle of *information hiding*. These attributes are therefore *private* or *protected* in the **System** class declaration. Without an insulated high-level structure it would be difficult to provide consistent, user-extendable modelling facilities. In other simulation systems (e.g. OMOLA), the high-level structure is protected by separating the modelling and development languages.

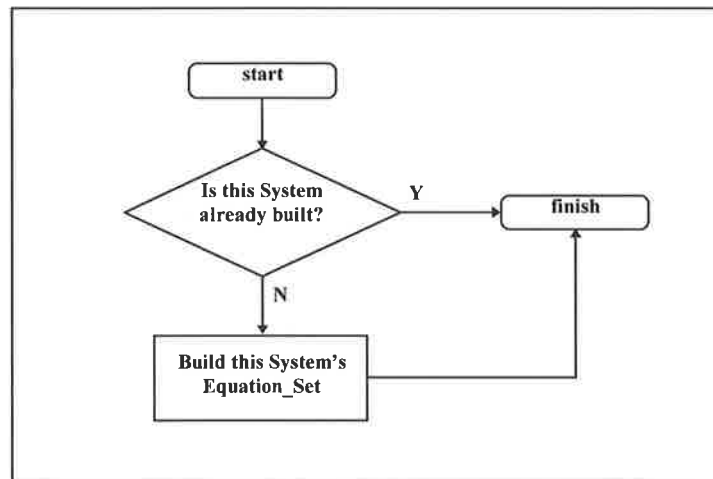
The executive interface functions are solely defined at the **System** level and are not virtual. This prevents a user-developer from redefining the functions at a lower level and corrupting the integrity of the data structure. The functions are designed to assign the executive-level pointers to specific objects at lower levels. The member interface functions are described in section A.1.1 of Appendix A.

Two pairs of virtual functions are defined for analysing the main steady-state and dynamic **Equation\_Set** objects in a **System**. One function in each pair performs the depth-first connection and analysis of the main **Equation\_Set** objects in each **System** and the second function performs the depth-first collection and building of **Vectors** of **Variable** and **Equation** pointers. The algorithms are shown below for steady-state analysis and collection/building in Figures 3.2 and 3.3. The dynamic algorithms are similar.



**Figure 3.2: System-class steady-state analysis algorithm.**

During the analysis the appended **Equation\_Set** objects become “extra” **Equation\_Set** objects that will be built as part of the **Equation\_Set** object that they are appended to (see section 3.4).



**Figure 3.3: System-class steady-state collection/building algorithm.**

The algorithms for analysis and building of **Equation\_Set**-types are described in section 3.4. The corresponding functionality is described in section A.1.2 of Appendix A.

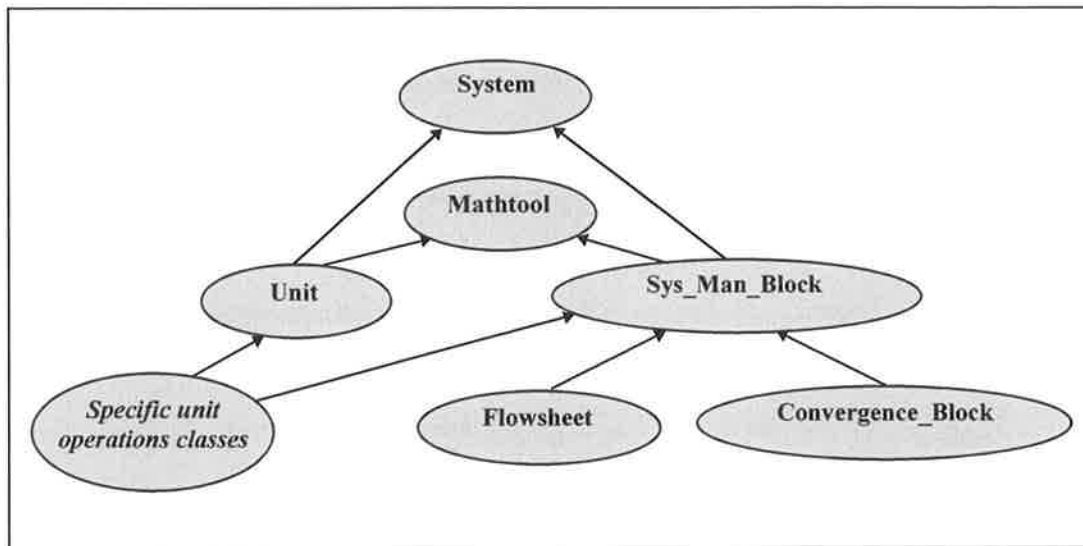
The **System** class also contains virtual functionality for unit models and associated ancillary functions. One steady-state and one dynamic function are provided for model definition. The functions are named `stst_model()` and `dynamic_model()` respectively. The default operation is to run the `dynamic_model()` function for steady-state solution. This is based on the assumption that a steady-state model is a dynamic model with time derivatives set to zero. The model developer is required to ensure that separate steady-state models can initialise a dynamic model correctly.

At the **System** level the `dynamic_model()` function does nothing and must be redefined for specific model types. These functions take no arguments because it is assumed that all the relevant model values are part of the class definition. If this is not the case then the two model functions can drive other functions that take arguments. Both the model functions return an integer value. This value is zero for a failed model evaluation and nonzero for a successful evaluation.

Three ancillary *virtual* functions are provided that take no arguments. The first is named `ss_output()` and is designed to be redefined for each unit class to output relevant solution **Variable** values at the conclusion of a steady-state simulation. The second function is named

`update()` and is automatically run for each unit after each time step in a dynamic simulation. It can be employed for updating past array values, model switches and variable output files, *etc.* The third function is supplied by both **System** and the **Mathtool** classes, named `disc_check()`. In lower level classes that inherit from **System** and **Mathtool**, the two functions simply merge. The function `disc_check()` is used for checking ahead for discontinuities in dynamic simulation. The function returns zero if no discontinuity exists over the next integration step and one if there is a discontinuity. The user is required to specify how a discontinuity is detected within individual unit models. An associated discontinuity variable is also available for the user to set or solve for the time at which the discontinuity occurred. The default behaviour is to return with no discontinuity.

The two main child classes of **System** (excluding specific unit operation classes) are **Unit** and **Sys\_Man\_Block**. **Flowsheet** and **Convergence\_Block** then inherit from the **Sys\_Man\_Block** class. Specific unit operation classes may inherit from any of **Unit**, **Flowsheet** or **Sys\_Man\_Block** as required. The **Mathtool** class is a joint parent of the **Unit** and **Sys\_Man\_Block** classes. The **Mathtool** class does not contribute to the process structure or unit models. It provides direct functionality to the **Unit** and **Sys\_Man\_Block** classes. The numerical methods are discussed in section 3.6. **Unit** is a pure interface class and contains no further structure or functionality from **System**. **Sys\_Man\_Block** contains no extra structure but has refined functionality for modelling, setting up and solving groups of connected **System**-types. The functionality drives the **System**-types it contains, for example the model and discontinuity functions. **Flowsheet** inherits directly from **Sys\_Man\_Block** and contains no extra structure or functionality. Note that the *public* functionality of the **System** class is available also. The inheritance tree is illustrated in Figure 3.4.



**Figure 3.4: Combined System/Mathtool class hierarchy.**

The **Sys\_Man\_Block** and **Flowsheet** classes contain a public *virtual* function named `initialise()` which sets initial estimates for solution **Variables** of each unit. Redefinition of the function in lower-level classes is optional.

The **Convergence\_Block** class is designed to reassign the input and output **Variables** associated with **Process\_Streams**. It contains its own **Variables**, **Equations**, an **Equation\_Set** and a **Vector** of torn **Process\_Streams**. There are *public* interface functions for tearing process streams, specifying solution methods and reassigning the input and output **Variables** of process units prior to solution. The interface functions are described in section A.1.3 of Appendix A:

### 3.3.2 Port Class and Descendants

The **Port** class hierarchy provides objects for connecting **System**-types together with **Stream**-types. The **Port** class contains a pointer to the **System**-type that owns it. This pointer is a *protected* class member. The **Port** class also contains generic *virtual* functionality for connecting input and output **Variables**. These functions are *public* because low-level **System**-types must be able to drive the connection functions.



A high-level interrogation function for accessing the **Variable** pointers in a low-level **Port**-type is provided. This function is *public*. This does not contravene the information hiding requirement of the high-level structure. The **Variable** pointers do not exist at the **Port** level; they only exist in lower level classes derived from **Port** (e.g. the **Process\_Input\_Port** class). The specific **Variable** objects assigned to the pointers in a low-level **Port** class are user-defined and therefore already accessible. The virtual interrogation function is designed to provide an interface for model debugging.

The **Port** class is divided into **Input\_Port** and **Output\_Port** classes. These two classes contain *protected* pointers to source and sink **Stream**-types. The high-level structure ends with the **Input\_Port** and **Output\_Port** classes. There is a direction associated with most connections in a flowsheet. The **Port** hierarchy caters for this with automatic functionality that instructs the **Variables** associated with an **Input\_Port**-type to remove themselves from an equation analysis. This is done because the inputs to a unit are usually outputs from somewhere else. For the purpose of constructing a solvable set of equations it is reasonable to assume that unit inputs are constant. If the inputs are actually solution variables they will be analysed as outputs of the preceding units. This automatic “switching off” is user-reversible.

The interface functions for the **Port**, **Input\_Port** and **Output\_Port** classes are described in section A.2.1 of Appendix A.

The executive-level **Port** classes are not employed as modelling objects. At the lower level six child classes are defined for process streams, signal streams and work streams with an input and an output class for each.

The **Process\_Input\_Port** and **Process\_Output\_Port** classes are for representing the entry and exit of process mixtures from a **System**-type. Both classes own a mixture composition, a total flowrate, and the temperature and pressure of the **System**-type that owns them. In addition the **Process\_Input\_Port** class knows the temperature and pressure of the **System**-type that feeds it and the **Process\_Output\_Port** class knows the temperature and pressure of the **System**-type that it is feeding. This structure provides bi-directional information flow. The application of this facility is demonstrated in Chapter 4. The flow, temperature and

pressure attributes are implemented as pointers to **Variables**. The mixture-composition attribute is a pointer to a **Vector**.

The **Variable** pointer attributes are *private* to the class, although member functions are provided to access them. Ideally the attributes should be completely inaccessible but this makes it difficult for user-developers to exploit the bi-directional information flow. The attributes are therefore *private* to ensure that access and modification is only possible through a deliberate member function call.

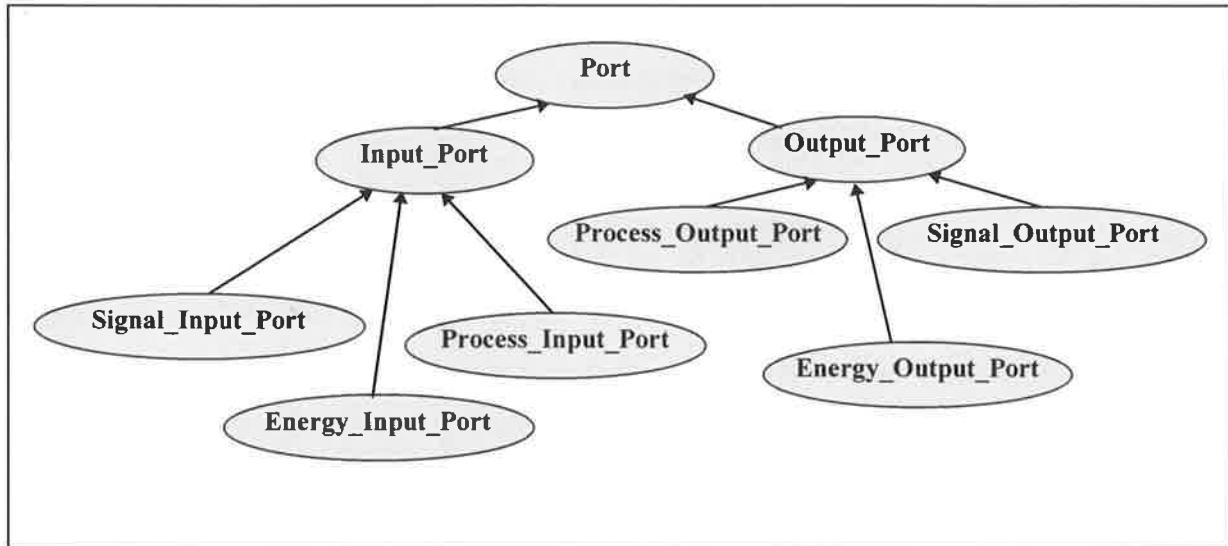
Each class has its own implementation of the `map()` and `get_vars(Vector<Variable*> &v)` functions of the **Port** class (see section A.2.1, Appendix A). The bi-directional temperature and pressure attributes are set automatically by the `map()` function. Interface functions are described in section A.2.2 of Appendix A.

The **Signal\_Input\_Port** and **Signal\_Output\_Port** classes are for representing the entry and exit of process signals from **System**-types, such as controller or measuring element signals. The classes only have one attribute, which is the signal. The class structures are exactly analogous to the **Process\_Input\_Port** and **Process\_Output\_Port** classes, except that bi-directional information flow is essentially automatic because there is no gradient associated with a signal. The interface functions for each class are described in section A.2.3 of Appendix A.

The **Energy\_Input\_Port** and **Energy\_Output\_Port** are classes for the transfer of energy to and from **System**-types. The type of work is not specific; it can be heat, shaft power or electrical *etc.* The class structure is the same as the **Signal\_Input\_Port** and **Signal\_Output\_Port** classes above. Connections with this **Port**-type have a nominal direction but do not affect the numerical analysis. The interface functions are described in section A.2.4 of Appendix A

It should be emphasised that a **Port**-type provides an *optional* connection point that is independent of any equation structure. It is not necessary to connect **Streams** (see 3.3.3 below) to all of the **Ports** in a **System**. For example, the **PI\_Controller** class owns a **Signal\_Input\_Port** for the setpoint. This enables cascaded control loops to be constructed.

However, a **PI\_Controller** may be used as a stand-alone controller without a connection to the setpoint **Variable** object. The complete **Port** class hierarchy is illustrated in Figure 3.5.

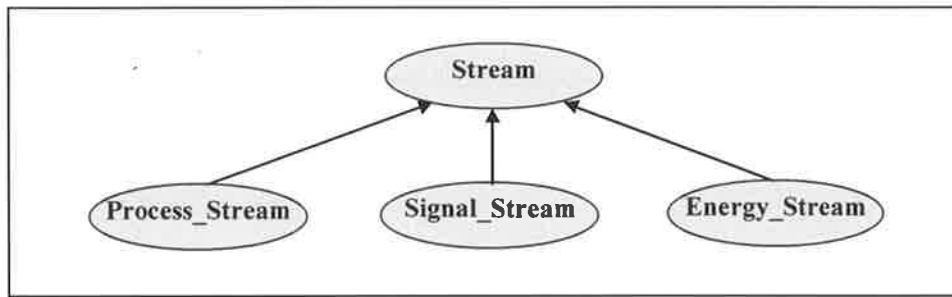


**Figure 3.5: Port class hierarchy.**

### 3.3.3 **Stream** class and Descendants

The basic **Stream** class is implemented as a pure connector with no functionality. It contains two *private* attributes for the source and sink **Ports** of the stream. It has three *friend* classes: **Port**, **Input\_Port** and **Output\_Port**. A class declared as a *friend* of another class in C++ is permitted to access the *private* and *protected* attributes of the class. This is reasonable in the case of the **Port** hierarchy, the connectivity is controlled by the **Ports** so a **Stream** object requires access to the physical mechanism of connection. The class has two interface functions, `get_source()` and `get_sink()`, described in section A.3.1 of Appendix A:

These two functions are used for obtaining access to the **Variable** objects associated with a particular **Stream**, for example with the **Convergence\_Block** class. There are three low-level child classes of **Stream** that correspond to the three low-level **Port**-types: **Process\_Stream**, **Signal\_Stream** and **Energy\_Stream**. None of these classes contain further structure or functionality. The three classes are derived to simplify coding of connections. The class hierarchy is illustrated in Figure 3.6.



**Figure 3.6: Stream class hierarchy.**

### **3.4 Mathematical Class Structure**

This section describes the implementation of the **Variable** and **Equation\_Set** hierarchies.

#### **3.4.1 Variable Class and Descendants**

The **Variable** class is used for representing possible solution variables in the simulator. It is the parent class for the **Derivative** and **Equation** classes. The **Variable** class contains a number of attributes. It contains a value and a pointer to another **Variable** for input-output connections, upper and lower bounds, a switch to determine if it is a solution variable or parameter and a switch to determine if it is to be analysed as part of an **Equation**. There are also attributes that determine if the **Variable** has been analysed and collected as part of an **Equation\_Set**. These attributes are *protected* and member functions are provided to assign or access the values of the attributes. The list of *public* member functions is reasonably large for this class (other member functions are *protected* for use by the **Equation\_Set** hierarchy which is a *friend* of this class). The *public* member functions are described in section A.4.1 of Appendix A. The most important member functions are named `operator()`, which returns the double precision value of the **Variable** object and `=`, for assigning values. Use of these functions is demonstrated later in this section.

The **Derivative** class also contains a connection to its state **Variable** object. This is a *protected* attribute and the **Dynamic\_Set** class is a friend of the class. The member functions are described in section A.4.2 of Appendix A.

The **Equation** class contains a list of the **Variable** objects that affect it. The list is modeller-defined. The **Equation** class is descended from the **Variable** class because the status functions, analysis and collection attributes and the value operator `()` are directly applicable.

The = operator is again overloaded for the class. The connection attribute of the **Variable** class is potentially useful for assigning a solution **Variable** to an **Equation**. The **Equation** class also contains two other protected attributes. One is a potential connection to a **Derivative** if the **Equation** is dynamic and the other is a connection to a **Variable**-based object if the **Equation** is to be written in explicit form. The *public* member functions of the class are explained below for convenience in examining the C++ example that follows.

`set_no_x(int n)` assigns the number of **Variables** that affect the **Equation**.

`include(Variable& v)` includes a **Variable** in the list.

`set_derivative(Derivative& d)` assigns the **Derivative** object of the **Equation**.

`set_exp_var(Variable& v)` assigns the explicit **Variable** for the **Equation**.

The principles of explicit and implicit **Equation** objects are best illustrated by example. Consider an equation written implicitly and explicitly below:

$$0 = x - xy \quad (3.1)$$

$$x = xy \quad (3.2)$$

In the class structure described, they would require an **Equation** object and two **Variable** objects (say, *e*, *x* and *y* respectively). Evaluation of the implicit form is straightforward. The C++ code would be:

```
e = x() - x()*y();
```

The terms `x()` and `y()` demonstrate the use of the `operator()` function described earlier. The explicit form is slightly more complex. Writing the code as

```
x = x()*y();
```

does not include the **Equation** object  $e$ . It also overwrites the value of the **Variable**  $x$  which is undesirable for interchangeable numerical methods. The solution is to use the **Equation** object itself as the left-hand-side of the expression, thus:

```
e = x()*y();
```

The code required to setup this equation is simple:

```
Equation e;//declare Equation object
Variable x,y;//declare Variable objects

e.set_no_x(2);//Equation is affected by 2 Variables
e.include(x);//Include the first Variable
e.include(y);//Include the second Variable
e.set_exp_var(x);//Set explicit Variable for the Equation
```

The **Equation** object then contains the result of the explicit calculation  $e = x() * y()$  and it also knows which **Variable** the explicit expression refers to. Therefore the implicit equation form is immediately available from the **Equation** object and the original value of  $x$  is preserved. The implicit equation residual can be calculated by the numerical methods without user intervention. The same principle is applicable to explicit and implicit dynamic **Equations**. The **Variable** hierarchy is drawn in Figure 3.7 below.

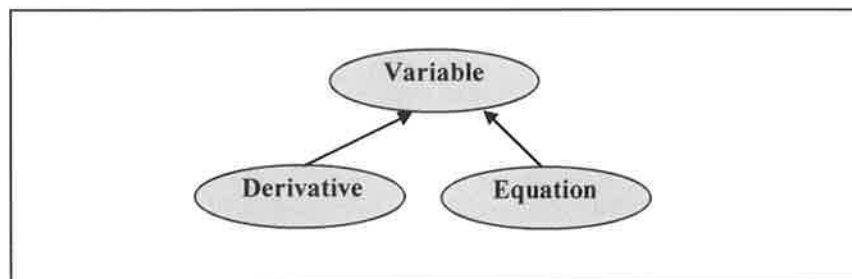


Figure 3.7: Variable class hierarchy.

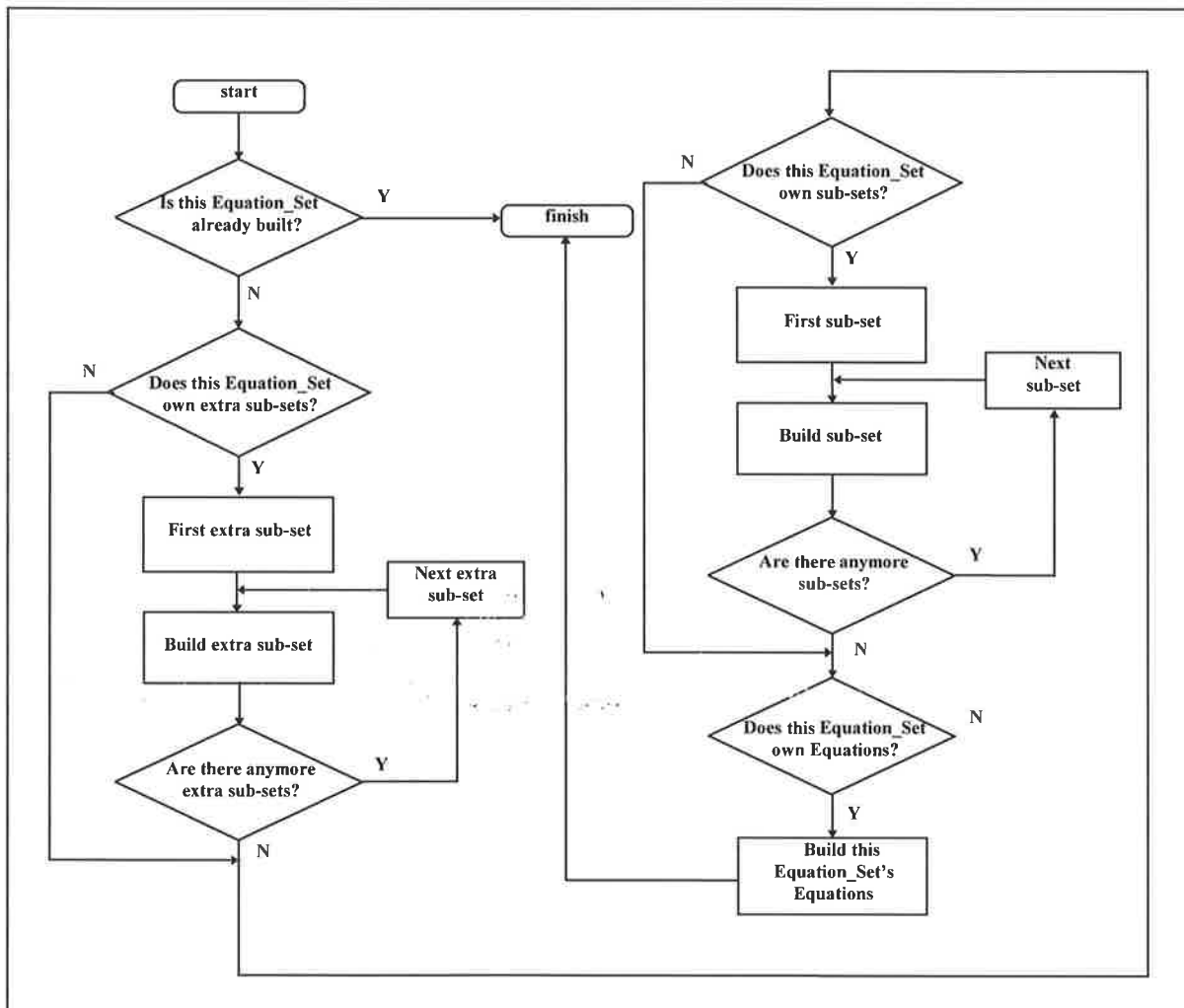
### 3.4.2 **Equation\_Set** and **Dynamic\_Set** classes

The **Equation\_Set** and **Dynamic\_Set** class structures are complex although the inheritance is simple. The **Dynamic\_Set** class inherits from **Equation\_Set**. An **Equation\_Set** owns a pointer to a **Vector** containing **Equation** objects. A single **Equation** object may only be incorporated as a single-element **Vector**. An **Equation\_Set** may only connect to one **Vector** of **Equations**. This one-to-one correspondence is enforced to encourage the isolation of specific sets of **Equations** within a **System**-type. A **Vector** of **Equations** may be connected to several **Equation\_Sets**. This permits the definition of different mathematical structures within a model. This is demonstrated in the next chapter. The design of the **Equation\_Set** hierarchy encourages the connection of **Equation\_Set** types so the one-to-one correspondence does not restrict mathematical modelling. The **Dynamic\_Set** class can further connect to other **Dynamic\_Set** or **Equation\_Set** objects. Mixed dynamic and steady-state **Equations** are not permitted within the **Vector** of **Equations** for a **Dynamic\_Set**. The **Vectors** are *protected* in the **Equation\_Set** class (to allow the **Dynamic\_Set** class to use them) and *private* in the **Dynamic\_Set** class. If a purely algebraic **Equation\_Set** is required for a dynamic simulation, it must be made a subset of an empty **Dynamic\_Set**.

The **Equation\_Set** class owns two **Vectors** of pointers to **Equations** and **Variables**. The **Dynamic\_Set** class owns five additional **Vectors** of pointers: one each for the set of **Derivatives**, algebraic **Variables** and **Equations** and the dynamic **Variables** and **Equations**. These **Vectors** are passed to the numerical methods for solution. The **Dynamic\_Set** class also contains a pointer to an independent **Variable**. The **Equation\_Set** class owns two **Vectors** of pointers to other **Equation\_Set** types. One **Vector** is for user-defined subsets of equations and the other is for extra subsets connected during the depth-first analysis. An example is a **Flowsheet** object: it has no user-defined subsets but collects the **Equation\_Sets** of the **System**-types it contains. A similar pair of **Vectors** is defined in the **Dynamic\_Set** class for dynamic subsets. The **Vectors** of pointers are *protected* in the **Equation\_Set** class and *private* in the **Dynamic\_Set** class. The interface functions for the classes are described in section A.4.4 of Appendix A.

The classes contain other functionality to ensure that **Equation\_Set** objects are not repeated in another **Equation\_Set**-type (i.e. structurally singular) and for analysing and building the **Vectors** of **Derivatives**, **Variables** and **Equations**. The owner **System**-type drives this

functionality. The building algorithms are illustrated for both classes in Figures 3.8 and 3.9. The analysis algorithms are similar, except that the “build” steps are replaced with “analysis” steps and the “extra” sub-sets do not exist. The “extra” sub-sets only exist after an analysis step, as described in section 3.3.1 and Chapter 2, section 2.3.1.



**Figure 3.8: Equation\_Set building algorithm.**

The analysis and collection functions are not virtual, so if a **Dynamic\_Set** is incorporated as the main steady-state set for a **System-type**; the **Derivatives** in the **Equations** are ignored. The state **Variables** become potential solution **Variables** of the steady-state set. This applies at all levels in a connected set of **Systems** and **Equation\_Sets** and so **Dynamic\_Sets** and **Equation\_Sets** can be mixed together in a steady-state analysis.



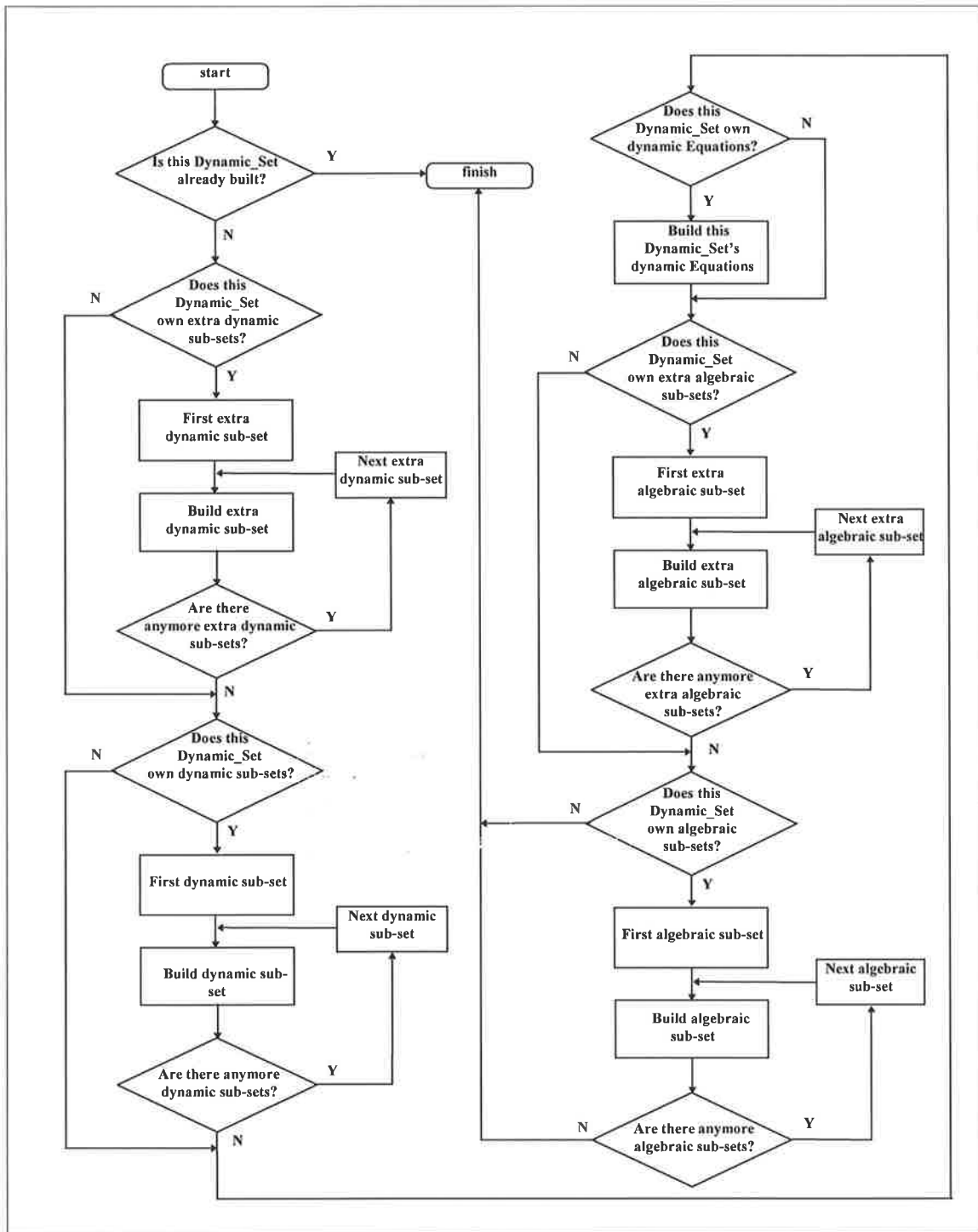


Figure 3.9: Dynamic\_Set building algorithm.

To illustrate the application of the mathematical structure classes, consider a simple model of balancing liquid height and volumetric flow in and out of a cylindrical tank. Two simultaneous equations can describe the system:

$$A \frac{dh}{dt} - F_{in} - F_{out} = 0 \quad (3.3)$$

$$F_{out} - C\sqrt{h} = 0 \quad (3.4)$$

Representation of these equations is simple:

```

Variable Fin,Fout,h,C;//declare Variables etc.
Derivative dhdt;
Vector<Equation> de(1,1),ae(1,1);//Single elements
Dynamic_Set d;
Equation_Set e;

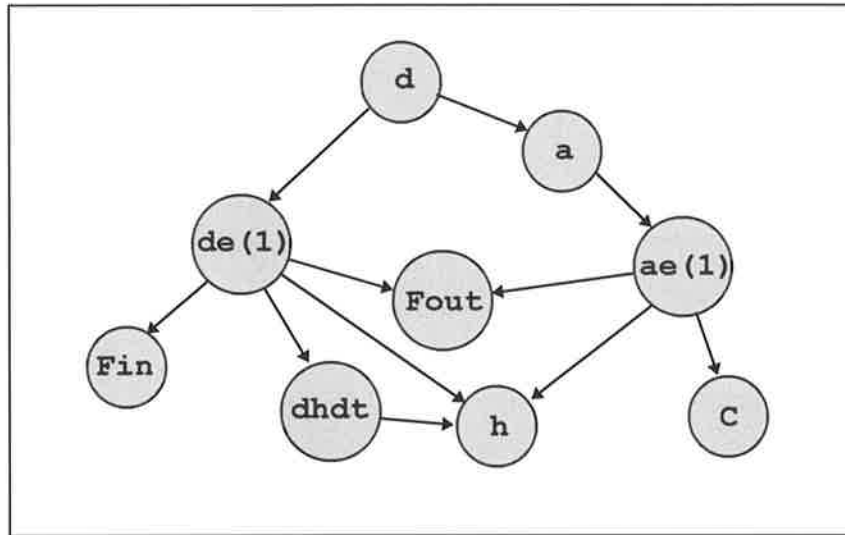
de(1).set_no_x(3);//3 Variables affect this Equation
de(1).include(h);
de(1).include(Fin);
de(1).include(Fout);
de(1).set_derivative(dhdt);//set the Derivative object
dhdt.set_state(h);

ae(1).set_no_x(3); //3 Variables affect this Equation
ae(1).include(h);
ae(1).include(Fout);
ae(1).include(C);

d.incorp_eqns(de);//include the de Vector
d.set_no_ae_sets(1);//has 1 algebraic set...
d.incorp_ae_set(ae,1);//... which is the ae object
//the set is now ready for analysis
//or Variable specification, e.g. constant(), var()

```

The code creates a connected tree of **Equation\_Set**, **Equation** and **Variable** objects, as illustrated in Figure 3.10.

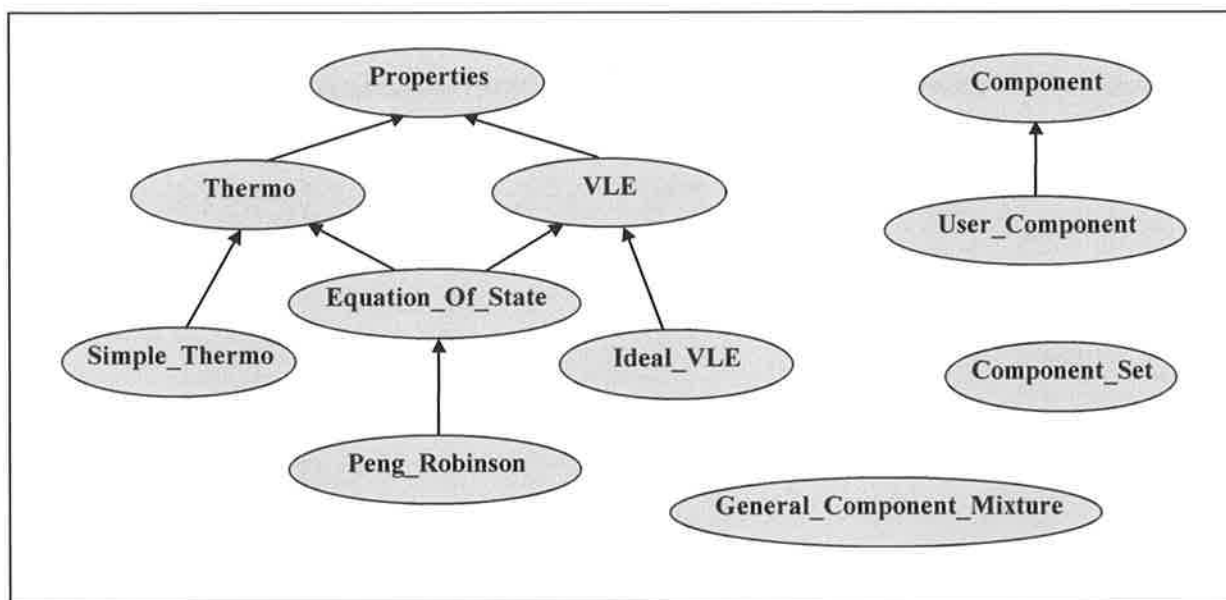


**Figure 3.10: Tank volume balance equation tree.**

The tree may be traversed and analysed for solution either as a dynamic or steady-state system. There is a redundant branch associated with the **Derivative**  $dhdt$  and **Variable**  $h$ . The redundant branch is compulsory in mathematical structure definition to ensure that the model developer caters for steady-state and dynamic analysis explicitly. The flow coefficient  $C$  is included as a **Variable** because in steady-state solution, a specified liquid height  $h$  permits exact calculation of the flow coefficient. Alternatively, specification of the coefficient permits calculation of the steady-state height. An example of a multi-component liquid mixing tank model is provided in the next chapter.

### **3.5 Component, General\_Component\_Mixture and Properties classes**

This section describes the **Component**, **General\_Component\_Mixture** and **Properties** hierarchies. The calculations are based on S.I. units in all cases. The classes contain comparatively little structure and the discussion emphasises the interface functions for various calculations. The physical property class hierarchies are illustrated in Figure 3.11.



**Figure 3.11: Physical property class hierarchies.**

### 3.5.1 Component class and Descendants

The attributes of a conventional chemical component were discussed in Chapter 2. The attributes of the **Component** class are: name, molecular weight, critical temperature, critical pressure, critical volume, boiling point at standard conditions, freezing point at standard conditions, acentric factor, dipole moment, liquid and vapour specific heat, liquid and vapour density and enthalpy of vapourisation at standard conditions. The class also contains a pointer to text file which stores these attributes based on the name of the **Component**. These are all *private* attributes. The attributes are defined in S.I. units where applicable. There are *public* member functions to access the values of the attributes, set the datafile and retrieve the **Component** information. The functions are described in section A.5.1 in Appendix A.

The class is refined into a **User\_Component** class. The class contains extra functionality to permit the user to define the properties, described in section A.5.2 of Appendix A.

A **Component\_Set** class is also defined. It provides a container for groups of **Components** to be attached to the **General\_Component\_Mixture** and **Properties** classes. The functionality is simple and is described in section A.5.3 of Appendix A.

### 3.5.2 General\_Component\_Mixture class

The **General\_Component\_Mixture** class contains a pointer to a **Component\_Set** and a corresponding double-precision **Vector** of mole fractions. It also contains pointers to **Thermo** and **VLE** objects and reference values for temperature and pressure. These attributes are *private*. The **Thermo** class is part of the **Properties** hierarchy and is discussed in the next section. The class contains *public* interface functions to calculate general properties of interest. Properties are calculated with respect to the reference values and state of the mixture. The member functions are described in section A.5.4 of Appendix A.

Most of the functionality reflects the capabilities of an equation of state, discussed in the next section. Some functionality will be invalid for simple approximate calculations. All vapour pressure calculations are based on the Antoine equation. While departure functions of the **Equation\_Of\_State** hierarchy were implemented, only the methods for enthalpy and specific heat are currently available.

### 3.5.3 Properties class and Descendants

The **Properties** class is a very simple parent for specific types of property calculation. The main attribute is a **Vector** of **Components**, which it accesses from the **General\_Component\_Mixture** it is attached to. **Properties** is the parent class for two types of property calculation: thermodynamics and vapour-liquid equilibrium. The classes are named **Thermo** and **VLE** respectively. These classes are high-level parents and cannot be employed as modelling objects. They contain virtual functionality for the calculation of thermodynamic and vapour-liquid equilibrium properties. Property calculation with lower-level **Properties**-types is accessed through the interfaces of the **General\_Component\_Mixture** class and not through the **Properties** descendants themselves.

The **Simple\_Thermo** class inherits from the **Thermo** class. The class employs extremely simple calculation methods, based on mole fraction averages of each **Component**'s properties. The **Ideal\_VLE** class provides simple vapour-liquid equilibrium calculations, based on Antoine constants. The user must supply the values of the constants, through the interface functions. The interface functions are described in section A.5.5 of Appendix A. Property calculation must be performed with the **General\_Component\_Mixture** class.

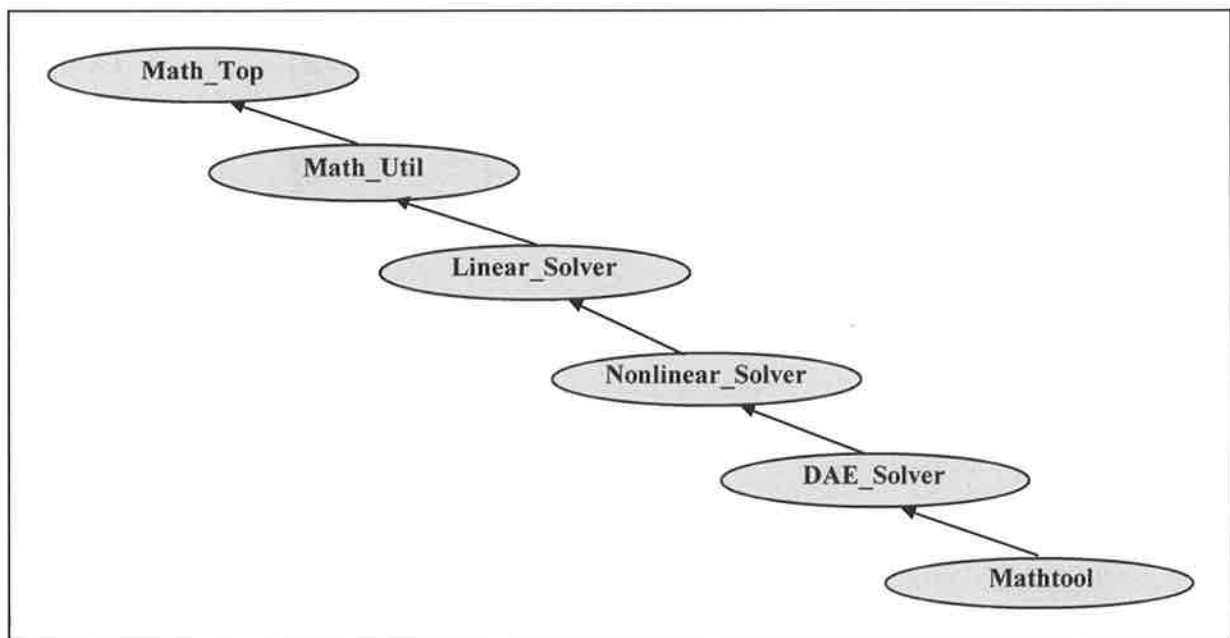
The **Equation\_Of\_State** class inherits from the **Thermo** and **VLE** classes. It is designed for cubic equations of state of the form:

$$P = \frac{RT}{V-b} - \frac{a}{V^2 + ubV + wb^2} \quad (3.5)$$

The class contains functionality for the calculation of fugacity coefficients for vapour-liquid equilibrium and departure functions for enthalpy, entropy, Helmholtz energy and Gibbs energy. The class inherits into a more specific **Peng\_Robinson** class that automatically specifies the values of  $a, b, u$  and  $w$  above. Reid (1988) discusses equations of state and calculation methods in detail.

### 3.6 Numerical Method Classes

Development of new algorithms was not an objective of the project. The basic numerical method classes were described in the previous chapter. The inheritance tree is presented below in Figure 3.12.



**Figure 3.12: Mathematical inheritance tree.**

The numerical methods are part of the high-level structure of the simulator and are not intended for modification. The Broyden and Newton methods are object-oriented modifications of code presented in Press et. al. (1992), designed to exploit the **Variable** and **Equation** structure. The Direct Substitution, Wegstein and Marquadt methods were coded from algorithms presented in Henley and Rosen (1969). The Backward Difference integrator was coded from an algorithm presented in Hall and Watt (1976). The backward differences are computed with a Vandermonde matrix (Press et. al. 1992).

The **Mathtool** class interface functions are described in section A.6 of Appendix A.

### **3.7 Summary**

The C++ class structure has been developed from the design requirements determined in Chapter 2. The class structure has been discussed in terms of the attributes and functionality available for a user to model and simulate unit operations and flowsheets. The process structure is modelled with refined classes from the **System**, **Port** and **Stream** hierarchies. Various connection types and multi-level **System**-based models can be created. The mathematical structure is modelled with classes from the **Variable** and **Equation\_Set** hierarchies. Multi-level mathematical structures are possible through a connection philosophy similar to the **System** hierarchy. **Equations** can be implicit or explicit. Model evaluation is based on steady-state and dynamic **System**-based model functions. Physical and mathematical representation is achieved with a relatively small set of attributes and functionality. The class hierarchies for **Component**, **General\_Component\_Mixture** and **Properties** were described with reference to the property methods available. Examples of the application of the C++ classes are provided in the next chapter.

# CHAPTER 4

## Modelling and Simulation

This chapter discusses some decomposition techniques applicable to process modelling, followed by examples demonstrating the M.O.P.S. C++ classes for modelling and simulation.

### 4.1 Decomposition Techniques

The final goal of a simulation model is a valid mathematical description of the process. The mathematical description can be developed in many different ways, depending on the frame of reference that is applied to the system under consideration. The most basic model development process is to code the complete set of equations for a model as one block within the framework of the modelling environment. For small process models, with say, less than ten equations this is reasonably fast and easy. There are drawbacks to this approach. Modification of the model is a complex process because a small change alters the entire model. This partially negates the advantages of object-oriented modelling. For large models single equation blocks are time-consuming and unwieldy. Likewise, model validation is also more difficult with large equation sets.

Usually a model is decomposed to reduce it to components that can be modelled and tested individually. Nilsson (1993) provides a detailed discussion of two model decomposition techniques, called Medium and Machine Decomposition and Primitive Behaviour Decomposition.

#### 4.1.1 Medium and Machine Decomposition

This technique divides the model into two systems, one for the machine-based model and one for the medium. The machine is the vessel or physical container of the unit operation. The medium is the mixture within the vessel. The model characteristics in each system depend on the frame of reference. The machine could be considered as owning the dynamic behaviour (holdups *etc.*) of the system and the medium could own the static behaviour of the system, for example chemical equilibrium. This is known as a static medium model. Alternatively the



machine and medium division can be based on intensive and extensive properties. Extensive properties such as total mass are part of the machine model and the intensive properties such as component concentration are part of the medium model. This is known as a dynamic medium model.

In both cases the two systems communicate via a connection interface. The interface for a mixture of components would be similar to the **Process\_Port** class described earlier. A consistent interface offers the potential for different machine models to be connected to different medium models. Machine and medium classes could be implemented with the **System** class structure.

Medium and machine decomposition can be provided in two ways in an object-oriented environment. A model could be constructed from medium and machine objects, or by inheriting the desired medium and machine classes into a new model class. Both methods require some sort of connection between common variables, for example the total input flow which affects the total mass balance in a machine model and the component concentrations in a medium model.

The emphasis of this project is to model complex systems as connected objects. However, C++ offers the facility to access the attributes of the individual parent classes and so decomposition by multiple inheritance is available. The **General\_Component\_Mixture** hierarchy provides some of the facilities of the static medium model.

#### 4.1.2 Primitive Behaviour Decomposition

A primitive behaviour of a model is behaviour that results from a particular modelling principle or assumption, such as conservation of mass. Primitive behaviour decomposition involves breaking a model up into compartments that define the model from the underlying modelling assumptions. If the compartments can be defined as modelling entities themselves with their own inputs, outputs and behaviour, modelling becomes a knowledge-based procedure. Model equation sets are constructed by selecting an appropriate modelling assumption, such as an object from a hypothetical **Mass\_Balance** class. Several primitive objects could be connected together to create a general model, with more specific details (*e.g.* reaction rates) coded in on an as-needed basis.

As stated previously, a knowledge-based implementation was considered outside the scope of the project. However, the existing class structure can accommodate aspects of a knowledge-based environment. Consider the definition of primitive behaviour classes for mass, component and energy balances. The mass and component balances could be incorporated into a **System**-based class that owned **Equation**, **Variable** and **Port**-type objects. The energy balance could be part of another **System**-based class with the same attributes plus an **Energy\_Output\_Port** and a **General\_Component\_Mixture**-type for thermodynamic calculations. In a specific reactor class these two primitives could be declared as objects of type **MassBal** and **EnergyBal**, say, with the reaction kinetics coded by the developer. Connections between the primitives and the rest of the model could be accomplished via the **Variable** class' connectivity or with **Stream**- and **Port**-types. The number of inputs and outputs would be parameters of the constructor functions for **MassBal** and **EnergyBal**. The basic equation structure for the model is defined from pre-validated modelling objects. A knowledge-based implementation is a very basic building-block approach to process modelling and simulation. The declaration of a flash-type for example is only a collection of modelling assumptions about mass, energy and phase equilibrium.

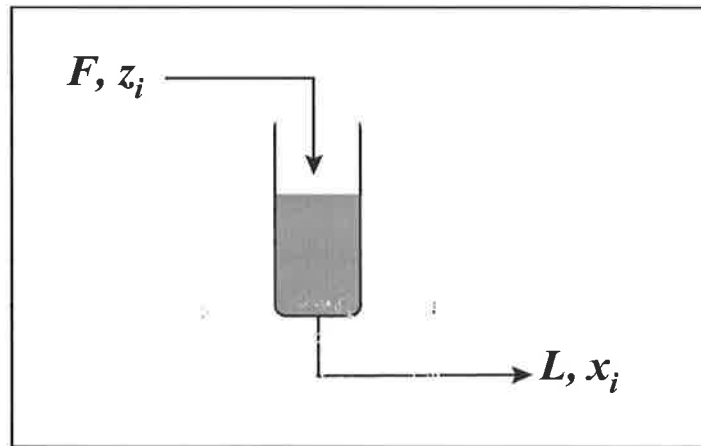
#### 4.1.3 Mathematical Decomposition

Methods have been described for the decomposition of a process model by various modelling principles and assumptions. A further decomposition is based on the mathematical structure of the modelling equations. This decomposition is applicable independently or as an extension to the decompositions above. It is partially derived from the **Equation\_Set** structure. The requirements and attributes of mathematical modelling with the **Equation\_Set** hierarchy are reviewed below:

- An **Equation\_Set** may connect to (or contain) any number of other **Equation\_Sets** as subsets.
- A **Dynamic\_Set** may connect to (or contain) any number of **Dynamic\_Sets** and/or **Equation\_Sets** as subsets.
- An **Equation\_Set**-type may contain only one **Vector** of **Equations**.
- A **Vector** of **Equations** in an **Equation\_Set** type may only contain either steady-state or dynamic equations, not a mixture.

- An algebraic set of **Equations** for a dynamic simulation must be part of an **Equation\_Set**. This **Equation\_Set** must then be a subset of a **Dynamic\_Set**. A **Dynamic\_Set** may contain subsets only and no dynamic **Equations** of its own.

The one-to-one correspondence between an **Equation\_Set**-type and a **Vector** of **Equations** suggests decomposing the mathematical model into groups related to a particular model aspect. This is best demonstrated with an example. Consider an open liquid mixing tank similar to the one discussed in Chapter 2, redrawn in Figure 4.1. The system equations are derived from a mole balance.



**Figure 4.1: Cylindrical liquid mixing tank.**

The equations and variables for the dynamic mole balance are given below:

$$\frac{dN_i}{dt} = Fz_i - Lx_i \quad 1 \dots n \quad (4.1)$$

$$N_i = x_i N_t \quad 1 \dots n \quad (4.2)$$

$$\sum_{i=1}^n x_i = 1 \quad (4.3)$$

$$\frac{N_t}{A\rho_{molar}} = h \quad (4.4)$$

$$P = \rho_{mass}gh \quad (4.5)$$

$$P_{out} = P + P_{atm} \quad (4.6)$$

$$\text{Variables: } x_i(n), z_i(n), N_i(n), N_t, L, F, h, P, P_{out}$$

The tank is adequately modelled by equations (4.1 - 4.6). A mole balance is performed because this is compatible with the rest of the simulator structure. The minimum number of **Vectors of Equations** is two, one for the dynamic equation (4.1) and one for the algebraic equations (4.2 - 4.6). This requires a corresponding **Dynamic\_Set** with an **Equation\_Set** connected as a subset to represent the DAE system. The decomposition could be extended further. Equation (4.2) relates component holdups to total holdup and could form their own **Equation\_Set** object. Equations (4.3), (4.4), (4.5) and (4.6) could be part of a second **Equation\_Set** object.

Depending on the simulation requirements, the system could be decomposed by separating the equations according to the requirements of steady-state and dynamic simulation. The full set could be solved in steady-state to initialise a dynamic simulation with the correct holdups. Equations (4.1) and (4.3) are adequate for a purely steady-state analysis where holdups are irrelevant.

The **Equation\_Set** hierarchy can accommodate a variety of equation structures within one model. Four **Equation\_Sets** and one **Dynamic\_Set** can cater for the options discussed so far. To do this, one **Equation\_Set** is attached to each of equations (4.2), (4.3) and the triplet (4.4),(4.5),(4.6). One **Dynamic\_Set** and one **Equation\_Set** is attached to equations (4.1). This is permitted within the data structure; a **Vector of Equations** may have a one-to-many connection with **Equation\_Sets** (the **Vector** does not “know” if it is attached to anything). The three **Equation\_Sets** for equations (4.2), (4.3), (4.4), (4.5) and (4.6) are then made algebraic subsets of the **Dynamic\_Set**. This provides the full steady-state and dynamic holdup model. The smaller steady-state model is created by connecting the **Equation\_Set** that owns equations (4.1) to the **Equation\_Set** for (4.3). The simulated model is selected by assigning the appropriate set through the `incorp_main_ss_set(Equation_Set& e)` and `incorp_main_dyn_set(Dynamic_Set& d)` functions of the **System**-class.

## 4.2 Modelling Examples

The mixing tank will be used as an initial example of modelling with the C++ classes. The model will be in a basic form without physical properties or an energy balance. The mathematical structure will be the full steady-state and dynamic model discussed above. Other models will be developed in this section to demonstrate bi-directional information flow,

connected system modelling, multiple inheritance of characteristics and complex physical property, energy and mass balance modelling. The modelling process employed is similar to the development of the simulator class structure; the physical attributes are determined followed by the mathematical structure. **Variable** and **Equation\_Set** types and most member functions are generally *public* attributes. This follows from a similar philosophy to the ASCEND (Piela et al. 1990) modelling language where the user can manipulate whatever element of the structure they desire. This facilitates model debugging and the initialisation and assignment of values. Unless specified, measurement units are in the S.I. system.

#### 4.2.1 Mixing Tank

The mixing tank drawn in Figure 4.1 has one input and one output stream. These connections are represented by a **Process\_Input\_Port** and a **Process\_Output\_Port**. Associated with each **Port**-type is a **Vector** of compositions, a temperature and a pressure. It is assumed for this simple example that the liquid level is not an interface **Variable** and that the vessel cannot overflow. The set of equations (4.1) - (4.6) contains the compositions, holdups, holdup derivatives, the tank level, area, fluid density and outlet pressure. The equations are assigned to two **Equation\_Sets** and one **Dynamic\_Set**: The **Dynamic\_Set** owns equation (4.1). One **Equation\_Set** owns equation (4.2) and equations (4.3) - (4.6) are allocated to the other **Equation\_Set**. The class declaration describes what the attributes of the **Tank** class will be and any functionality that the class owns. The class must inherit the high-level **System** attributes from the **Unit** class. The C++ class declaration is given below. Comment statements are indicated by italic text that follows a double forward slash (*//*). C++ code is in bold type.

```
#ifndef MIXTANK_HPP
#define MIXTANK_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "mixtank.hpp" .
#include "unit.hpp" //Includes the Unit class header file.
//The unit.hpp header file contains all the basic header files
//required for modelling units, such as mathematical
//operations, screen output, Equation_Sets, Variables etc.

class Mixing_Tank:public Unit{//Commence class declaration.
    //Inherits from the Unit class.
    public://Attributes at this modelling level are public.
```

```

Vector<Variable> z,x,N;//Declare inlet and outlet
//composition and holdup vectors.
Vector<Derivative> dNdt;//Declare holdup derivative
//vector.
Vector<Equation> de,ae1,ae2;//Declare dynamic and
//algebraic equation vectors.
Variable Pin,Pout,Tin,Tout;//Declare inlet and
//outlet pressure and temperature.
Variable F,L,Ntot,h,P,Patm;//Declare inlet and
//outlet flow, total molar holdup,
//liquid height, height pressure
//and atmospheric pressure.
Equation_Set eqnset1,eqnset2;//Declare
//Equation_Sets.
Dynamic_Set dynset1;//declare Dynamic_Set
double rhomass,rhomolar,Area,g;
//Declare mass density,
//molar density, tank area and
//gravitational acceleration.
//Not solution variables. Must be set
//by the user.
int nc;//Declare variable to hold number of
//components in the tank.
Process_Input_Port in;//Declare input port.
Process_Output_Port out;//Declare output port.

int dynamic_model();//Declare the dynamic model.
void setup();//Declare function to connect ports.
Mixing_Tank(int n, int no_comps);//Declare
//constructor function. Both arguments
//are integer types. Argument n is an
//index number for the System-type,
//no_comps is the number of components
//in a stream.
~Mixing_Tank(){};//Declare destructor function.
//It does nothing but it is good
//programming practice to include a
//null definition
};//End of class declaration

#endif//End of #ifndef above

```

It should be noted that no assignments are permitted within a class declaration in C++; hence the **Vector** objects are declared with no dimension. A molar and mass density are required. Normally these are available directly from the physical property classes but this model has no physical property calculation objects. The **Vector** dimensions are allocated within the class constructor. The three member functions `dynamic_model()`, `setup()` and

Mixing\_Tank(int n, int no\_comps) must be defined for the class. The definitions are given below. The constructor function is the most complex.

```
Mixing_Tank::Mixing_Tank(int n, int no_comps):Unit(n){//Start
    //constructor definition.
//The constructor is where an object of the class is put
//together. Note that the 'n' argument is passed directly to
//the Unit constructor function, Unit(n).

    int i;//Declare counter variable i.


    nc=no_comps;//Assign value in no_comps to nc.

    //Allocate Vector objects.
    z.build(1,nc);
    x.build(1,nc);
    N.build(1,nc);
    dNdt.build(1,nc);
    de.build(1,nc);
    ae1.build(1,nc);
    ae2.build(1,4);
    //Finished allocating Vector objects.

    //Start physical structure definition.
    in.set_tot_flow(F);//Assign the total flow Variable for
    //the inlet.
    in.set_fracs(z);//Assign the composition Vector for the
    //inlet.
    in.set_press_inlet(Pin);//Assign the pressure Variable
    //for the inlet.
    in.set_temp_inlet(Tin); //Assign the temperature Variable
    //for the inlet.
    in.set_press_owner(Patm);//Assign the owner's pressure
    //for the inlet. Necessary for
    //bi-directional information
    //flow. Open tank will always be
    //at atmospheric pressure.
    in.set_temp_owner(Tout);//Assign the owner's temperature
    //for the inlet. Necessary for
    //bi-directional information
    //flow.

    out.set_tot_flow(L);//Assign the total flow Variable for
    //the outlet.

    out.set_fracs(x);//Assign the composition Vector for the
    //outlet.
```



```

out.set_press_outlet(Pout); //Assign the pressure Variable
    //for the outlet.
out.set_temp_outlet(Tout); //Assign the temperature
    //Variable for the outlet.

//This unit has no energy balance and is an open tank.
//Therefore the temperature inlet and outlet Variables
//should be connected for continuity from one unit to the
//next.

Tout.connect_to(Tin); //Connect outlet temperature to the
    //inlet temperature.

set_no_inpstrms(1); //Set the number of input streams.
set_no_outstrms(1); //Set the number of output streams.

own_input_port(in,1); //The tank owns the inlet.
own_output_port(out,1); //The tank owns the outlet.
//End physical structure definition.

//Start mathematical structure definition.
for(i=1;i<=nc;i++){ //Loop over dynamic Equations.
    //Corresponds to equations (4.1) in the
    //text.
    de(i).set_no_x(5); //Each differential Equation has 5
        //Variables.
    de(i).include(F); //Include inlet flow F.
    de(i).include(z(i)); //Include inlet composition
        //element z(i).
    de(i).include(L); //Include outlet flow L.
    de(i).include(x(i)); //Include outlet composition
        //element x(i).
    de(i).include(N(i)); //Also include the state
        //Variable (holdup element N(i)).
    de(i).set_derivative(dNdt(i)); //Set the derivative
        //for this Equation.
    de(i).set_exp_var(dNdt(i)); //The Equation stores
        //the Derivative dNdt(i) as
        //its explicit Variable
        //(see Chapter 3, section 3.4.1).
} //End dynamic equation loop.

for(i=1;i<=nc;i++){ //Loop over Derivatives.
    dNdt(i).set_state(N(i)); //assign the state Variable
        //for each Derivative.
} //End loop over Derivatives.

for(i=1;i<=nc;i++){ //Loop over first set of algebraic
    //Equations. Corresponds to equations (4.2)
    //in the text

```



```

ae1(i).set_no_x(3); //Each algebraic Equation in this
//Vector has 3 Variables.
ae1(i).include(N(i)); //Include holdup N(i).
ae1(i).include(x(i)); //Include composition element
//x(i).
ae1(i).include(Ntot); //Include total holdup Ntot.
//Note that no explicit Variable is assigned for
//these Equations. They must be written in the
//model in fully equation-oriented form.
} //End loop over first set of algebraic Equations.

ae2(1).set_no_x(nc); //Corresponds to equation (4.3) in the
//text. Has the nc elements of the x
//composition Vector.
for(i=1;i<=nc;i++){ //Loop over the x composition
//elements.
ae2(1).include(x(i));
//No explicit Variable is assigned.
}

ae2(2).set_no_x(2); //Corresponds to equation (4.4) in the
//text. Has 2 Variables.
ae2(2).include(Ntot); //Include the total holdup Ntot.
ae2(2).include(h); //Include liquid height h.
//No explicit Variable is assigned.

ae2(3).set_no_x(2); //Corresponds to equation (4.5) in the
//text. Has 2 Variables.
ae2(3).include(P); //Include the height pressure P.
ae2(3).include(h); //Include the liquid height h.
//No explicit Variable is assigned.

ae2(4).set_no_x(2); //Corresponds to equation (4.6) in the
//text. Has 2 Variables. Note that
//Patm is not included in this Equation
//because it is never a solution
//Variable.
ae2(4).include(Pout); //Include the outlet pressure Pout.
ae2(4).include(P); //Include the height pressure P.
//No explicit Variable is assigned.

dynset1.incorp_eqns(de); //Attach the de Vector to the
//Dynamic_Set object dynset1.

eqnset1.incorp_eqns(ae1); //Attach the ae1 Vector to the
//Equation_Set eqnset1.

eqnset2.incorp_eqns(ae2); //Attach the ae2 Vector to the
//Equation_Set eqnset2.

```

```

dynset1.set_no_ae_sets(2); //dynset1 has two algebraic
                          //subsets.
dynset1.incorp_ae_set(eqnsset1); //eqnsset1 is a subset.
dynset1.incorp_ae_set(eqnsset2); //so is eqnsset2.

incorp_main_ss_set(dynset1); //dynset1 is the main
                             //steady state set for this
                             //model. Dynamic_Sets can be
                             //analysed for steady-state
                             //solution (see Chapter 3,
                             //section 3.4.2).
incorp_main_dyn_set(dynset1); //dynset1 is the main
                              //dynamic set for this model.

//End mathematical structure definition.

Patm = 1013.0; //Set atmospheric pressure in kPa.
g = 9.81; //Set gravitational acceleration in m/s/s.

} //End constructor definition.

```

The constructor describes an equation-oriented model. A very important aspect of equation-oriented simulation and modelling must be emphasised here. If the unit computation order is arbitrary, any interface **Variable** (i.e. a **Variable** associated with a **Port-type**) that is a potential solution **Variable** must be part of a simultaneous **Equation**. Consider equations (4.4), (4.5) and (4.6). These are very simple equations with obvious explicit solutions. If the equations were in explicit form and a unit downstream of the tank is calculated before the tank, the value of the pressure in the connecting stream will be incorrect. The stream pressure will still be at the value from the previous iteration until the tank is calculated. The convergence of a steady-state system could be retarded or destabilised and a dynamic integration would contain a constant error. At this stage the project has no algorithm available to determine computation order. An ordering algorithm would be a great advantage because the requirement of including all interface **Variables** in simultaneous **Equations** increases the dimensions of a problem. Alternatively a sparse solution method could be provided to reduce the solution time of the fully equation-oriented system.

The code for evaluation of the model is placed inside the virtual `dynamic_model()` function:

```

int Mixing_Tank::dynamic_model(){//Start mathematical model
                                //definition

    int i;//Declare counter.
    double xsum;//Declare mole fraction summation.
    //Evaluate DEs.
    for(i=1;i<=nc;i++){
        de(i) = F()*z(i)() - L()*x(i)();
        //Note that the derivative does not appear
        //in the evaluation. This is because the
        //Derivative has been set as the explicit
        //Variable for the Equation. See constructor
        //above and Chapter 3, section 3.4.1.
    }//End DE evaluation.

    xsum=0.0;//Initialise mole fraction summation.
    //Evaluate first set of AEs.
    for(i=1;i<=nc;i++){
        ae1(i) = x(i)()*N(i)() - Ntot();
        //These Equations have no explicit Variable.
        xsum = xsum + x(i)();//Calculate summation.
    }//End first AE evaluation.

    //Evaluate other AEs.
    ae2(1) = xsum - 1.0;
    ae2(2) = h() - Ntot()/(rhomolar*A);
    ae2(3) = rhomass*g*h() - P()*1000.0;//1000.0 converts to
                                        //kPa.
    ae2(4) = Pout() - (P() + Patm());
    //End other AE evaluation.

    return OK;//Model evaluation complete. Return OK signal.
}//End mathematical model definition

```

The `dynamic_model()` function above is run for both steady-state and dynamic simulation. The final member function definition is trivial and drives the `map()` functions of the **Port**-types within the tank:

```

void Mixing_Tank::setup(){//Start setup/connection function
                            //definition.

    in.map();//Connect inlet.
    out.map();//Connect outlet.

}//End setup/connection function definition.

```

The class declaration and function definitions are straightforward and the purpose of the code statements is reasonably clear. For many models, a class definition and these three member functions are sufficient to describe a model that is compatible and solvable within the simulator structure. The tank area, mass and molar density must be set before the model can be simulated. In this example, no member functions have been defined to set these parameters, so the parameters must be accessed directly from an object of the **Mixing\_Tank** class. The tank only has one input and one output stream. The **System** class provides basic connectivity functions (Chapter 3, section 3.3.1), which enable **Process\_Streams** to be connected to a **Mixing\_Tank** object. An example is provided below that demonstrates connectivity and specification of the class' parameters:

```
Mixing_Tank tank1(99,1); //index number 99, 1 component
Process_Stream strm1, strm2;
...
...//etc
tank1.rhomass = 1000.0; //e.g. water, 1000 kg/m^3
tank1.rhomolar = 55.55; //kgmol/m^3
tank1.area = 0.785; //about 1 metre diameter
...
...//etc
tank1.inp_stream(strm1,1);
tank1.out_stream(strm2,1);
```

Non-descriptive connectivity is acceptable for **System**-types with few connections. For models with several connections, it is usually necessary to define specific functions within the model class that connect **Stream** objects appropriately. This is demonstrated in the examples later in this chapter.

#### 4.2.2 Bi-Directional Information Flow

Bi-directional information flow will be demonstrated with a **Control\_Valve** class. As discussed in Chapter 2, flow through a valve can be modelled with a square-root dependence on the pressure drop. For a control valve, flow  $F$  is given by equation (4.7) below. The dependence of flow on mixture density is ignored in this example.  $P_{in}$  and  $P_{out}$  are the inlet and outlet pressures of the valve,  $x$  is the stem position and  $C$  is the valve constant.

$$F_{out} = xC\sqrt{P_{in} - P_{out}} \quad (4.7)$$

The inlet and outlet pressures of the valve are defined by the vessels that are upstream and downstream of the valve. The inlet pressure is automatically available from the inlet **Port**-type but the outlet pressure must be obtained from the downstream vessel. In addition, the flow through the valve dictates the flow out of the upstream vessel for an incompressible fluid. The upstream vessel's outlet flow should therefore be reassigned to the valve's outlet flow. A valid connection structure is still created without the flow **Variable** reassignment. In this case the upstream vessel and not the valve will own the solution **Variable** for flow. The operations required to reassign connections are moderately complicated but are explained in detail below. The flow will be reassigned in this example.

The **Control\_Valve** class has three connection points: a **Signal\_Input\_Port**, a **Process\_Input\_Port** and a **Process\_Output\_Port**. It also has flow, stem position, valve constant, temperature and pressure **Variables**, an **Equation\_Set** and a **Dynamic\_Set**. A composition **Vector** is also required. The class declaration is given below.

```

#ifndef CONVALVE_HPP
#define CONVALVE_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is " convalve.hpp" .
#include " unit.hpp" //Includes the Unit class header file.
//The unit.hpp header file contains all the basic header files
//required for modelling units, such as mathematical
//operations, screen output, Equation_Sets, Variables etc.

class Control_Valve:public Unit{//Commence class declaration.
    //Inherits from the Unit class.
public://Attributes at this modelling level are public.

    Vector<Variable> z;//Declare composition vector.
    Vector<Equation> ae;//Declare algebraic equation
        //vector.
    Variable Pin,Pout,T;//Declare inlet and
        //outlet pressure and a temperature.
    Variable Fout,Fin,C,x;//Declare flows, valve
        //constant and position variable.
    Equation_Set eqnset1;//Declare Equation_Set.
    Dynamic_Set emptydyn;//Declare Dynamic_Set. Will
        //not have any dynamic Equations.
    int nc;//Declare variable to hold number of
        //components in composition Vector.
    Process_Input_Port in;//Declare input port.
    Process_Output_Port out;//Declare output port.

```

```

Signal_Input_Port sig_in;//Declare signal port.

//Declare 3 connection functions.
void flow_in(Stream& str);
void flow_out(Stream& str);
void signal_in(Stream& str);

int dynamic_model();//Declare the dynamic model.
void setup();//Declare function to connect ports.
Control_Valve(int n, int no_comps);//Declare
    //constructor function. Both arguments
    //are integer types. Argument n is an
    //index number for the System-type,
    //no_comps is the number of components
    //in a stream.
~Control_Valve(){};//Declare destructor function.
    //Does nothing but it is good
    //programming practice to include a
    //null definition
};//End of class declaration

#endif//End of #ifndef above

```

The class' constructor is:

```

Control_Valve::Control_Valve(int n,
    int no_comps):Unit(n){//Start
    //constructor definition.
//The constructor is where an object of the class is put
//together. Note that the 'n' argument is passed directly to
//the Unit constructor function, Unit(n).

    nc=no_comps;//Assign value in no_comps to nc.

    //Allocate Vector objects.
    z.build(1,nc);
    ae.build(1,1);//Vector only has one Equation.
    //Finished allocating Vector objects.

    //Start physical structure definition.
    in.set_tot_flow(Fin);//Assign the total flow Variable for
        //the inlet.
    in.set_fracs(z);//Assign the composition Vector for the
        //inlet.
    in.set_press_inlet(Pin);//Assign the pressure Variable
        //for the inlet.
    in.set_temp_inlet(T); //Assign the temperature Variable
        //for the inlet.
    out.set_tot_flow(Fout);//Assign the total flow Variable

```

```

        //for the outlet.
out.set_frac(z); //Assign the composition Vector for the
        //outlet.
out.set_press_outlet(Pout); //Assign the pressure Variable
        //for the outlet.
out.set_temp_outlet(T); //Assign the temperature
        //Variable for the outlet.

//Note that the composition Vector x and
//temperature T are common to both the inlet and outlet
//process ports. There is no holdup or energy balance in
//the valve, so common Variables provide direct inlet -
//outlet connections.

sig_in.set_signal_var(x); //Assign the signal Variable
        //for the signal inlet

set_no_inpstrms(2); //Set the number of input streams.
//One process input and one signal input.
set_no_outstrms(1); //Set the number of output streams.

own_input_port(in,1); //The valve owns the process inlet.
//See definition of "flow_in(Stream& str)" below.

own_input_port(sig_in,2); //The valve owns the signal
        //inlet.
//See definition of "signal_in(Stream& str)" below.

own_output_port(out,1); //The tank owns the process
        //outlet.
//See definition of "flow_out(Stream& str)" below.

//End physical structure definition.

//Start mathematical structure definition.

ae(1).set_no_x(4); //Corresponds to equation (4.7) in the
        //text.

ae(1).include(Fout); //Include the outlet flow Fout.
ae(1).include(C); //Include the valve constant C.
ae(1).include(x); //Include the stem position x.
ae(1).include(Pin); //Include the inlet pressure Pin.
ae(1).include(Pout); //Include the outlet pressure Pout.

eqnset1.incorp_eqns(ae); //Attach the ae Vector to the
        //Equation_Set eqnset1.

emptydyn.set_no_ae_sets(1); //emptydyn has one algebraic
        //subset.

```

```

emptydyn.incorp_ae_set(eqnset1); //eqnset1 is a subset.
//Note that emptydyn owns no Equations itself.

incorp_main_ss_set(emptydyn); //emptydyn is the main
//steady state set for this
//model. Dynamic_Sets can be
//analysed for steady-state
//solution (see Chapter 3,
//section 3.4.2). Could also
//have assigned eqnset1 here.
incorp_main_dyn_set(emptydyn); //emptydyn is the main
//dynamic set for this model.

//End mathematical structure definition.

} //End constructor definition.

```

The model function is trivial:

```

int Control_Valve::dynamic_model() { //Start mathematical model
//definition

    ae(1) = Fout() - x()*C()*sqrt(Pin() - Pout());
    return OK;

} //End mathematical model definition

```

Three connection functions were declared in the class. These functions call the **System**-level connection functions. Application-specific connection functions should be defined in classes with complex connectivity to clarify the purpose of each input and output **Stream**-type. The index *n* in the call to the `out_stream(Stream& strm, int n)` should correspond to the index *n* in the call to `own_output_port(Output_Port& p, int n)`. The same applies to the input functions. This is demonstrated as follows:

```

void Control_Valve::flow_in(Stream& str) { //Attaches inlet
//Process_Stream

    inp_stream(str,1); //The Process_Input_Port "in" is the
//first "owned" Input_Port-type in the
//constructor function.
}

```



```

void Control_Valve::signal_in(Stream& str){//Attaches inlet
                                     //Signal_Stream

    inp_stream(str,2);//The Signal_Input_Port " sig_in" is
    the
                                     //second " owned" Input_Port-type in
                                     //the constructor function.
}

void Control_Valve::flow_out(Stream& str){//Attaches outlet
                                     //Process_Stream

    out_stream(str,1);//The Process_Output_Port " out" is the
    //first " owned" Output_Port-type in
    //the constructor function.
}

```

The setup() function is where the bi-directional information flow is exploited. The downstream pressure is obtained and connected to by interrogating the **Process\_Output\_Port** out. The upstream vessel's flow is reassigned to the valve's outlet flow Fout. The setup() function is :

```

void Control_Valve::setup(){//Start setup/connection function
    //definition.

    in.map();//Connect process inlet Variables.
    sig_in.map();//Connect signal inlet Variable.
    out.map();//Connect process outlet Variable.
    //The connections have been made. Now the Variables
    //can be reassigned.

    //Reassign pressure.
    Pout.connect_to(out.get_pressure_sink());
    //The statement above is a dual function call. The
    //call " out.get_pressure_sink()" obtains the address
    //of the pressure Variable of the downstream vessel.
    //The valve's output pressure Variable, Pout, is then
    //immediately connected to the pressure Variable
    //returned by the get_pressure_sink()function.
    Pout.check(OFF);//Pout is a solution Variable
    //of the downstream vessel and should not be analysed
    //as part of the valve's Equation_Set.

    //Reassign flow. Uses another dual function call.
    (Fin.get_connection()->connect_to(Fout));
    //This is more complicated. The call
    //" Fin.get_connection()" obtains the address of the
    //outlet flow Variable of the upstream vessel. The "->"

```

```

//operator dereferences the address (obtains the actual
//Variable object)and connects the outlet flow Variable
//of the upstream vessel to the outlet flow Variable of
//the valve.
(Fin.get_connection())->check(OFF);//The upstream flow is
//now a solution Variable of the valve and should not be
//analysed as part of the upstream vessel's Equation_Set
};//End setup/connection function definition.

```

The reassignments above cannot be made until the **Port**-types and their **Variables** have been connected by the `map()` functions. The **Variable** `Fin` is a dummy that takes no part in solution or equation analysis. The purpose of `Fin` is to enable a connection to be made between the **Process\_Input\_Port** `in` and the **Process\_Output\_Port** of the upstream vessel prior to the connection reassignment. If the flows were not to be reassigned, `Fin` would not be required at all. The **Ports** `in` and `out` would both be attached to `Fout` in a similar fashion to the **Variable** `T`.

The reassignment would affect sequential-modular simulation. Without the reassignment, the flow solution **Variable** is owned by the upstream vessel. Therefore in a sequential-modular simulation, the valve could not be solved for the flow because the flow would be specified by the solution of the upstream vessel. With the reassignment, the upstream vessel could not be solved for the flow because the flow is a solution **Variable** of the valve. The ownership of the flow solution **Variable** affects the problem specifications that may be made. Similar problems occur with the pressure reassignment. The simulator data structure permits these problems to be overcome easily with the **Sys\_Man\_Block** class. The upstream vessel, valve and downstream vessel could be incorporated into a **Sys\_Man\_Block** object. Then the **Sys\_Man\_Block** object can be incorporated into a **Flowsheet** object instead of the three separate units. A **Convergence\_Block** object can then drive the **Flowsheet** object. The three-unit system can then be analysed and solved as one equation-oriented **System**-type, removing the restrictions on problem specification. The reassignment of **Variable** connections becomes irrelevant. The code for grouping the vessels together and incorporating them into a **Flowsheet** object is simple:

```

.....
.....
Process_Stream strm1, strm2; //Other Streams would be
                             //required
                             //in a large flowsheet
Vessel upstream(1,6), downstream(2,6);
Control_Valve v1(2,6);
Sys_Man_Block smb1(999);
Flowsheet flwsht(111);
.....
.....
upstream.out_stream(strm1,1);
v1.flow_in(strm1);
v1.flow_out(strm2);
downstream.inp_stream(strm2,1);
.....
.....
smb1.set_sys(3);
smb1.incorp_sys(upstream,1);
smb1.incorp_sys(v1,2);
smb1.incorp_sys(downstream,3);
.....
.....
flwsht.incorp_sys(smb1,1);
.....
.....

```

#### 4.2.3 Connected System Modelling

Flowsheet-level modelling with connected **Systems** is briefly introduced above. Connected **System** objects can also be created within a class definition. This is illustrated with the definition of a **Ratio\_Controller** class. The **Ratio\_Controller** class will be developed from simplified **PI\_Controller** and **Ratio\_Block** classes. The final model is an aggregation of an object from each class. The advantage of this approach is that the model is constructed completely from tested, validated objects in the same way as a flowsheet definition.

The mathematical model for the **PI\_Controller** class is:

$$\frac{dI(t)}{dt} = e(t) \quad (4.8)$$

$$e(t) = y_{sp}(t) - y_m(t) \quad (4.9)$$

$$c(t) = K_c \left( e(t) + \frac{1}{\tau_I} I(t) \right) + c_s \quad (4.10)$$

$K_c$  is the controller gain,  $\tau_i$  is the integral time,  $e(t)$  is the error,  $I(t)$  is the integral of the error,  $y_{sp}(t)$  is the setpoint,  $y_m(t)$  is the measured variable,  $c(t)$  is the controller output signal and  $c_s$  is the steady-state controller output or bias signal.

The mathematical model for the **Ratio\_Block** class is:

$$R_{out} = \frac{y_1}{y_2} \quad (4.11)$$

$R_{out}$  is the output ratio and  $y_1$  and  $y_2$  are the input signals.  $R_{out}$  becomes the measured variable  $y_m(t)$  in the **Ratio\_Controller** class and  $y_{sp}(t)$  becomes the specified signal ratio.

The **PI\_Controller** class declaration for this example is :

```
#ifndef PICONT_HPP
#define PICONT_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "picont.hpp" .
#include "unit.hpp" //include Unit class header file.

class PI_Controller::public Unit{
public:
    double Kc,Ti;//Gain and integral time.
    Variable c,cs,e,ysp,ym,I;//Variables in equations
    //4.8 - 4.10 above.
    Derivative dIdt;
    Vector<Equation> de,ae;//de is for equation 4.8,
    //ae is for equations 4.9 and 4.10.
    Dynamic_Set de_set,ae_set;//For de and ae above.
    Signal_Input_Port meas_in,sp_in;//Measured value
    //and setpoint signal input ports.
    Signal_Output_Port sig_out;//Controller signal
    //output port.
    void set_Kc(double k){Kc = k;};
    void set_Ti(double t){Ti = t;};

    //Connection functions below.
    void measured_in(Stream& str);
    void setpoint_in(Stream& str);
    void signal_out(Stream& str);

    void setup();
    int dynamic_model();
};
```

```

PI_Controller(int n); // "Normal" constructor.
//Argument n is set to the System parent's
//index number. Used for a stand-alone
//controller object.

PI_Controller(); //C++ default constructor.
//Same as above, with no index number set.
//Required for declaring objects within
//class structures, e.g. Ratio_Controller
//model with objects.
PI_Controller(int n, int m);
//Used for multiple-inheritance modelling.
//Only creates mathematical structures and attaches
//Variables to Port-types. Does not manipulate
//System-level data structure. See section 4.2.4
//below.
}; //End class declaration

#endif

```

This class is a simplification of the class actually implemented in the simulator and inherits directly from **Unit**. The `dynamic_model()` function for the **PI\_Controller** class is:

```

int PI_Controller::dynamic_model() { //start model

    de(1) = e(); //de(1) is the LHS of equation 4.8.
    ae(1) = ysp() - ym(); //ae(1) is the LHS of
//equation 4.9.
    ae(2) = Kc*(e() + 1.0/Ti*I()) + cs();
//ae(2) is the LHS of equation 4.10.
    return OK;
} //End model.

```

The `setup()` function for the **PI\_Controller** class is:

```

void PI_Controller::setup() { //Start setup.

    meas_in.map();
    sp_in.map();
    sig_out.map();

} //End setup.

```

The `Ratio_Block` class declaration is:

```
#ifndef RATBLOCK_HPP
#define RATBLOCK_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "ratblock.hpp" .
#include "unit.hpp" //include Unit class header file.
class Ratio_Block::public Unit{

    public:
        Variable Rout,y1,y2;//Variables in equation 4.11
        //above.
        Vector<Equation> ae;//ae is for equation 4.11.
        Dynamic_Set de_set,ae_set;//For ae above.
        Signal_Input_Port sig_A,sig_B;//Measured
        //values.
        Signal_Output_Port ratio;//Ratio output port.

        //Connection functions below.
        void signal_A_in(Stream& str);
        void signal_B_in(Stream& str);
        void ratio_out(Stream& str);
        //These connection functions are only valid
        //for a stand-alone Ratio_Block object.
        //They must be redefined in the child
        //Ratio_Controller class because the
        //connectivity is partly based on the
        //System-level data structure.

        void setup();
        int dynamic_model();
        Ratio_Block(int n);//" Normal" constructor.
        //Argument n is set to the System parent's
        //index number. Used for a stand-alone
        //object.
        Ratio_Block();//C++ default constructor.
        //Same as above, with no index number set.
        //Required for declaring objects within
        //class structures, e.g. Ratio_Controller
        //model with objects.

        Ratio_Block(int n, int m);
        //Used for multiple-inheritance modelling.
        //Only creates mathematical structures and attaches
        //Variables to Port-types. Does not manipulate
        //System-level data structure.
}; //End class declaration

#endif
```

The `dynamic_model()` function for the **Ratio\_Block** class is:

```
int Ratio_Block::dynamic_model(){//start model

    ae(1) = Rout() - y1()/y2();
    return OK;
} //End model.
```

The `setup()` function for the **Ratio\_Block** class is:

```
void Ratio_Block::setup(){//Start setup.

    sig_A.map();
    sig_B.map();
    ratio.map();

} //End setup.
```

The connection functions of the **PI\_Controller** class are:

`measured_in(Stream& str)` connects the **Stream** that supplies the measured variable.

`setpoint_in(Stream& str)` connects the **Stream** that supplies the setpoint in cascade control systems. This is an optional connection.

`signal_out(Stream& str)` connects the **Stream** that carries the output signal.

The connection functions of the **Ratio\_Block** class are:

`signal_A_in(Stream& str)` connects the **Stream** that supplies the first signal.

`signal_B_in(Stream& str)` connects the **Stream** that supplies the second signal.

`ratio_out(Signal& str)` connects the **Stream** that carries the output ratio.

The `Sys_Man_Block` class is a suitable parent because it is designed for managing sets of connected `Systems`. No extra `Variables` are required for the model definition. A `Dynamic_Set` is required for the main steady-state and dynamic set of the class. The class definition is below:

```

#ifndef RATIOCON_HPP
#define RATIOCON_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "ratiocon.hpp" .
#include "smblock.hpp" //Includes the Sys_Man_Block class
//header file.

#include "picont.hpp" //Also include PI_Controller
#include "ratblock.hpp" //and Ratio_Block class headers

class Ratio_Controller:public Sys_Man_Block{ //Commence class
//declaration. Inherits from the
//Sys_Man_Block class.

    public: //Attributes at this modelling level are public.

        PI_Controller ctrl1r; //Declare PI_Controller
//object
        Ratio_Block ratio; //Declare Ratio_Block object
        Signal_Stream internal_conn; //Declare a
//Signal_Stream for an internal
//connection.
        Dynamic_Set dynset1; //Declare Dynamic_Set.

        //Declare 3 connection functions.
        void signal_A_in(Stream& str);
        void signal_B_in(Stream& str);
        void signal_out(Stream& str);

        Ratio_Controller(int n); //Declare constructor
//function.
        ~Ratio_Controller(){}; //Declare destructor
//function.
//Does nothing but it is good
//programming practice to include a
//null definition
}; //End of class declaration

#endif //End of #ifndef above

```



In the class definition, no `setup()` or `dynamic_model()` functions are declared. The **Sys\_Man\_Block** class runs the setup and model functions of any subsystems it contains. The user is required to explicitly indicate the subsystems. This is part of the constructor function below:

```
Ratio_Controller::Ratio_Controller(int n):Unit(n){//Start
//constructor definition.
//The constructor is where an object of the class is put
//together. Note that the 'n' argument is passed directly to
//the Unit constructor function, Unit(n).

    set_sys(2);//Initialise number of subsystems
    incorp_sys(ratio,1);//Incorporate subsystems
    incorp_sys(ctrllr,2);

    //Start internal connections. Creates a pseudo-
    //flowsheet.
    ratio.ratio_out(internal_conn);
    ctrllr.measured_in(internal_conn);
    //End internal connections

    incorp_main_ss_set(dynset1);//dynset1 is the main
//steady-state set for this
//model.
    incorp_main_dyn_set(dynset1);//dynset1 is the main
//dynamic set for this model.

} //End constructor definition.
```

The simulator executive automatically attaches the **Dynamic\_Sets** of the `ratio` and `ctrllr` objects to `dynset1` in the equation analysis step. The two objects do not have to be connected with a **Signal\_Stream**. They may be connected directly with their respective **Variable** objects for measured variable and output ratio because these attributes are *public*. This breaks the boundaries defined by the **Port**-types within the objects. However, there might be circumstances where a **Port**-type is not available for the **Variable** objects of interest and direct connection between **Variable** objects is the only option.

If a class contains its own **Equations** and its own **Ports** in addition to other **Systems**, new `setup()` and `dynamic_model()` functions are required. The functions must operate on the **Equations** and **Ports** of the class and the subsystems are operated on by the functionality in the ancestor **Sys\_Man\_Block** class.

The three connection functions declared in the class definition drive the connection functions of the `ratio` and `ctrlr` objects:

```
void Ratio_Controller::signal_A_in(Stream& str){
    ratio.signal_A_in(str);//Connect directly to ratio
                               //object
}

void Ratio_Controller::signal_B_in(Stream& str){
    ratio.signal_B_in(str);//Connect directly to ratio
                               //object.
}

void Ratio_Controller::setpoint_in(Stream& str){
    ctrlr.setpoint_in(str);//Connect directly to ctrlr
                               //object.
}

void Ratio_Controller::signal_out(Stream& str){
    ctrlr.signal_out(str);//Connect directly to ctrlr
                               //object.
}
```

An example of a `Ratio_Controller` object is provided below:

```
Ratio_Controller RC(1);//index number 1.
Signal_Stream signal_B,signal_A,con_sig;
... //etc.
RC.signal_A_in(signal_A);
RC.signal_B_in(signal_B);
RC.signal_out(con_sig);
//Note no setpoint signal.
```

#### 4.2.4 Multiple Inheritance Modelling

Multiple inheritance modelling is performed at the class level as opposed to the aggregation approach with objects in the previous section. Multiple inheritance modelling does not provide the same level of automatic consistency as aggregation. The developer models with inherited attributes that do not necessarily provide an encapsulated object with specific interfaces. Depending on the application this can be considered an advantage or a disadvantage.

The **System** data structure supports permit multiple inheritance of characteristics as described in section 4.1.1. However, the successful implementation of multiple-inheritance modelling is dependent on the constructor functions of the parent classes. In the modelling examples above, the constructors drive a number of **System**-level functions to define **System** boundaries with **Ports** and incorporate **Equation\_Sets**. The **System** class is a *virtual base class* (Ellis and Stroustrup, 1994) of **Unit** and **Sys\_Man\_Block** and consequently only one copy of the **System** data structure exists for each model class. This is necessary for the principle of incorporated subsystems to work effectively. The virtual base class also restricts the operations that may be carried out on the **System** data structure. A class with multiple parents (or multiple levels of refinement, e.g. **System-Unit-Controller-PI\_Controller**) will have several ancestor constructor functions that are run prior to the constructor function of the new class. If all the low-level constructors of a class with multiple parents try to manipulate the same **System**-level data structure in turn, an object of the new class will not initialise correctly (if at all). The solution is straightforward: any model class capable of being a parent of another class must contain at least two constructor functions. The constructor functions require different arguments in order for the correct constructor to be identifiable and run. One of the constructor functions will be similar to those already presented, with mathematical and physical connection structures fully described. The other constructor will still describe the mathematical structures and associate **Variables** with **Port**-types, but will not declare ownership of **Port**-types or incorporate **Equation\_Set**-types into the **System**-level structure. This constructor only operates on modelling objects at each class' level and ignores the **System**-level data structure. The reason for this is because it is only at the most refined level (i.e. the new model class) that the final structure is known. Therefore the most refined constructor is the only one that may operate on the **System**-level structure. This constructor will be quite complex as a result. All the ancestor **Equation\_Sets**, connection **Variables** and

**Ports** must be known and available to the constructor (*public* or *protected*) in order for it to build the model correctly. The low-level connection functions (e.g. `signal_out(Stream& str)` above) must be redefined for each parent class, because part of each connection is made at the **System** level. This is more complicated than aggregating a model with objects because it requires a great deal more knowledge of the ancestor modelling classes.

The **Ratio\_Controller** class will now be redefined as a class with multiple parents. The multiple-inheritance constructors of the parent classes must first be examined:

```
PI_Controller::PI_Controller(int n, int m):Unit(n){

    de.build(1,1); //Only 1 element, equation 4.8.
    ae.build(1,2); //Equations 4.9 and 4.10.

    de(1).set_no_x(1); //One variable affects equation 11.
    de(1).include(e);
    de(1).set_outputvar(dIdt); //Explicit output variable
    //for this equation is the derivative dIdt. See
    // "dynamic_model()" function in section 4.2.3.
    dIdt.set_state(I);

    de_set.incorp_eqns(de);

    ae(1).set_no_x(4); //Four variables affect equation 4.9.
    ae(1).include(e);
    ae(1).include(I);
    ae(1).include(c);
    ae(1).include(cs);
    ae(1).set_outputvar(c); //Explicit output variable
    //for this equation is the controller output c. See
    // "dynamic_model()" function in section 4.2.3.

    ae(2).set_no_x(3); //Three variables affect equation 4.10.
    ae(2).include(e);
    ae(2).include(ysp);
    ae(2).include(ym);
    ae(2).set_outputvar(e); //Explicit output variable
    //for this equation is the controller error e. See
    // "dynamic_model()" function in section 4.2.3.

    ae_set.incorp_eqns(ae);

    de_set.set_no_ae_sets(1);
```

```

    de_set.incorp_ae_set(ae_set);

    meas_in.set_signal_var(ym);
    sp_in.set_signal_var(ysp);
    sig_out.set_signal_var(c);

} //end constructor

```

In this constructor, the argument `int m` is not used. The extra argument is designed to be used for conditional construction (e.g. `if(m==1)` etc.) in complex multiple-inheritance models where different structures may be necessary depending on the modelling requirements.

The multiple-inheritance constructor for the **Ratio\_Block** class is:

```

Ratio_Block::Ratio_Block(int n, int m):Unit(n){

    ae.build(1,1); //Only 1 element, equation 4.11.

    ae(1).set_no_x(3); //3 variables affect equation 4.11.
    ae(1).include(Rout);
    ae(1).include(y1);
    ae(1).include(y2);
    //Fully implicit equation, no explicit variable.
    //See dynamic_model() function in section 4.2.3.
    ae_set.incorp_eqns(ae);

    de_set.set_no_ae_sets(1);
    de_set.incorp_ae_set(ae_set);

    sig_A.set_signal_var(y1);
    sig_B.set_signal_var(y2);
    ratio.set_signal_var(Rout);
} //End constructor

```

The class declaration for the multiple-parent **Ratio\_Controller** class is similar to the aggregated object class declaration:

```

#ifndef RATIOCON_HPP
#define RATIOCON_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "ratiocon.hpp" .

```

```

#include " picont.hpp" //Also include PI_Controller
#include " ratblock.hpp" //and Ratio_Block class headers

class Ratio_Controller:public PI_Controller, public
    Ratio_Block{//Commence class
    //declaration. Inherits from the
    //Sys_Man_Block class.
public://Attributes at this modelling level are public.

    Dynamic_Set dynset1;//Declare Dynamic_Set.

    //Declare 4 connection functions.
    void signal_A_in(Stream& str);
    void signal_B_in(Stream& str);
    void signal_out(Stream& str);
    void setpoint_in(Stream& str);
    //Only 4 Ports used out of 6.
    void setup();
    int dynamic_model();

    Ratio_Controller(int n);//Declare constructor
    //function.
    ~Ratio_Controller(){};//Declare destructor
    //function.
    //Does nothing but it is good
    //programming practice to include a
    //null definition
};//End of class declaration

#endif//End of #ifndef above

```

Some of the connection functions must be redefined to reflect the different **System**-level structure. There are now six **Ports** in the **Ratio\_Controller** class. Only four of these are required for external connections. The constructor for the **Ratio\_Controller** class must run (specifically) the multiple-inheritance constructors of the two parent classes. It must also explicitly combine the **Equation\_Sets** of the two parent classes together, allocate the various **Ports** to the **System** structure, connect **Variable** ysp to Rout and remove ysp from the mathematical structure, as follows:

```

Ratio_Controller::Ratio_Controller(int n):
    PI_Controller(n,0),Ratio_Block(n,0){//Start
    //constructor definition.
//The constructor is where an object of the class is put
//together. The Unit and System class structures are
//provided by the parent classes. Note the argument '0'
//in the call to the parent multiple-inheritance constructors.

```

```
//This argument is not used in either of the parents'  
//constructors, it could be set to any valid integer value.
```

```
incorp_main_ss_set(dynset1); //dynset1 is the main  
                             //steady-state set for this  
                             //model.  
incorp_main_dyn_set(dynset1); //dynset1 is the main  
                              //dynamic set for this model.  
  
yssp.connect_to(Rout); //Connect variables.  
yssp.check(OFF); //yssp is really Rout and not  
//a separate Variable anymore. It will be examined  
//for analysis and collection as Rout.  
  
set_no_inpstrms(3); //3 potential input connections.  
own_input_port(sig_A,1);  
own_input_port(sig_B,2);  
own_input_port(sp_in,3);  
  
set_no_outstrms(1); //1 output connection.  
own_output_port(sig_out);  
  
dynset1.set_no_dyn_subsets(2); //Has no Equations  
//of its own, just the Dynamic_Sets of the parent  
//classes.  
dynset1.incorp_dyn_set(PI_Controller::de_set);  
dynset1.incorp_dyn_set(Ratio_Block::de_set);  
//The ":" is the scope resolution operator. It  
//identifies a function or piece of data as an  
//attribute of a particular class.  
//i.e. PI_Controller::de_set means the object  
//de_set that is owned by the PI_Controller class.  
  
} //End constructor definition.
```

The redefined connection functions are:

```
void Ratio_Controller::signal_A_in(Stream& str){  
  
    inp_strm(str,1); //System-level.  
}  
  
void Ratio_Controller::signal_B_in(Stream& str){  
  
    inp_strm(str,2); //System-level.  
}
```

```

void Ratio_Controller::setpoint_in(Stream& str){
    inp_strm(str,3); //System-level.
}

void Ratio_Controller::signal_out(Stream& str){
    out_strm(str,1); //System-level.
}

```

New `setup()` and `dynamic_model()` functions must be defined to run the parent functions:

```

int Ratio_Controller::dynamic_model(){//start model

    PI_Controller::dynamic_model();//Run parent models.
    Ratio_Block::dynamic_model();
    return OK;

} //End model.

```

The `setup()` function for the **Ratio\_Block** class is:

```

void Ratio_Controller::setup(){//Start setup.

    PI_Controller::setup();//Run parent setups.
    Ratio_Block::setup();

} //End setup.

```

In the `setup()` function above, all six **Ports** of the two parent classes will be mapped. This is acceptable because connection to **Ports** is optional and so the two **Ports** without connections (`meas_in` from **PI\_Controller** and `ratio` from **Ratio\_Block**) will automatically ignore any connection commands. The `setup()` function could also have been redefined to map only the four **Ports** with potential connections.

The construction of the **Ratio\_Controller** class with multiple inheritance does not require much more code than the aggregated object example, but more detailed knowledge about the underlying class structures is required. Objects of the new class may also be corrupted easily.

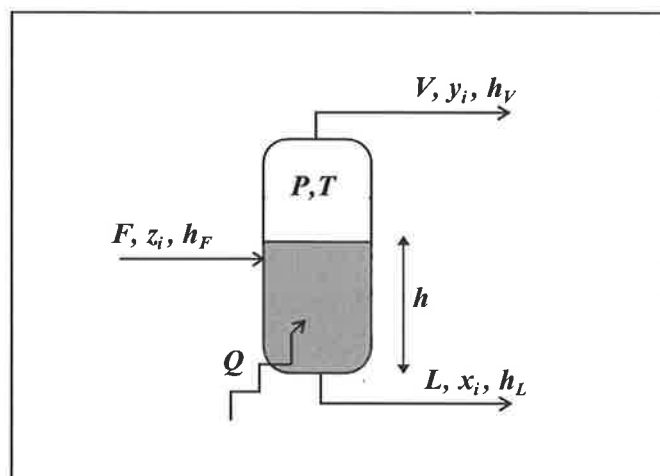


All the connection functions of the parent classes are *public* and therefore available from the child class through the scope resolution operator or directly (e.g. `Ratio_Block::signal_A_in(Stream& str)` and `measured_in(Stream& str)`). These connection functions still operate on the **System** structure at the parents' level and will corrupt the structure created by the **Ratio\_Controller** constructor unless they are redefined to do nothing. The connection functions of the internal objects in the aggregated **Ratio\_Controller** class are also available, but invoking them requires the identification of a specific internal object, i.e. `ctrlr` or `ratio`. The use of the multiple-inheritance based **Ratio\_Controller** class is exactly the same as the aggregated class.

In further examples all models are created by aggregation or a completely new class definition without multiple inheritance.

#### 4.2.5 Modelling with Physical Properties

As a final modelling example, a dynamic, molar holdup flash model will be developed with a two-phase component mixture, equation-of-state physical property calculations and an energy balance. The vessel is illustrated in Figure 4.2.



**Figure 4.2: Flash vessel.**

The equations for the dynamic molar holdup model of an equilibrium flash vessel with  $n$  components are given below.  $N_i$ ,  $N_{L_i}$  and  $N_{V_i}$  denote the total, liquid and vapour molar holdups respectively for each component.  $N_L$  and  $N_V$  denote the total molar holdups in the liquid and vapour phases respectively.  $V_{tot}$ ,  $V_V$  and  $V_L$  denote the total vessel, vapour and

liquid volumes. The model is directly applicable to steady-state simulation if the liquid level  $h$  is specified.

$$\frac{dN_i}{dt} = Fz_i - Lx_i - Vy_i \quad 1..n \quad (4.12)$$

$$(N_L C_{\rho L} + N_V C_{\rho V}) \frac{dT}{dt} = Q + Fh_F - Lh_L - Vh_V \quad 1 \quad (4.13)$$

$$0 = y_i - K_i x_i \quad 1..n \quad (4.14)$$

$$0 = \sum_{i=1}^n x_i - 1 \quad 1 \quad (4.15)$$

$$0 = \sum_{i=1}^n y_i - 1 \quad 1 \quad (4.16)$$

$$0 = N_i - N_{V_i} - N_{L_i} \quad 1..n \quad (4.17)$$

$$0 = N_L x_i - N_{L_i} \quad 1..n \quad (4.18)$$

$$0 = N_V y_i - N_{V_i} \quad 1..n \quad (4.19)$$

$$0 = V_{tot} - V_L - V_V \quad 1 \quad (4.20)$$

$$0 = V_L - \frac{N_L}{\rho_L} \quad 1 \quad (4.21)$$

$$0 = V_V - \frac{N_V}{\rho_V} \quad 1 \quad (4.22)$$

$$0 = h - \frac{V_L}{A} \quad 1 \quad (4.23)$$

**Total Equations:  $5n + 7$**

**Variables:**

$F, V, L, T, P, Q,$

$N_L, N_V, V_L, V_V, h,$

$z_i(n), x_i(n), y_i(n),$

$N_i(n), N_{L_i}(n), N_{V_i}(n) \quad \text{Total Variables: } 6n + 11$

In Figure 4.2 there are four obvious **Ports**: feed, liquid, vapour and heat duty. Three more **Ports** are required, for transmitting level, temperature and pressure signals. The constructor allocates equations (4.12) - (4.23) to separate **Equation\_Sets** according to their purpose. Equations (4.12) and (4.13) are dynamic equations (de and de\_set below), (4.14) are equilibrium relations (eqbm and eqbm\_set), (4.15) and (4.16) are mole fraction summations (mfs and mfs\_set), (4.17) are total component mole balances (cmb and cmb\_set), (4.18) are liquid phase mole balances (lmb and lmb\_set), (4.19) are vapour phase mole balances (vmb and vmb\_set) and (4.20) - (4.22) are volume balances (vb and vb\_set). The vessel is assumed to be cylindrical. The vessel area and volume are also set

through the constructor. The constructor function for the **Flash** class is listed in Appendix B. The class declaration and dynamic model are listed below:

```

#ifndef FLASH_HPP
#define FLASH_HPP
//Standard C/C++ practice for header files, avoids multiple
//inclusion of files. File name is "ratiocon.hpp" .
#include "unit.hpp" //include Unit class header file.
#include "physprop.hpp" //Includes the physical properties
//header files.

class Flash:public Unit{
    public:
        double Vol,Area,hmax;
        int nc;//Number of components.
        Vector<double> K;//Equilibrium constants.
        Vector<Variable> x,y,z,N,Nv,Nl;//Liquid, vapour
        //and feed compositions, total molar, vapour and
        //liquid holdups.
        Vector<Derivative> dNdt;//Molar holdup derivatives.
        Derivative dTdt;//Temperature derivative.
        Variable Tin,Pin,T,P,Q,F,V,L,VL,VV,NL,NV,h;//Other
        //variables.
        Vector<Equation> de,eqbm,mfs,cmb,lmb,vmb,vb;
        Dynamic_Set de_set;
        Equation_Set eqbm_set,mfs_set,cmb_set,
            lmb_set,vmb_set,vb_set;
        //de = dynamic equations (eqns 8 & 9).
        //eqbm = equilibrium equations (eqn 10).
        //mfs = mole fraction summation (eqns 11 & 12).
        //cmb = component mole balance (eqn 13).
        //lmb = liquid mole balance (eqn 14).
        //vmb = vapour mole balance (eqn 15).
        //vb = volume balances (eqn 16,17,18 & 19).
        Peng_Robinson peng_rob;//Physical properties.
        General_Component_Mixture VL_mix;//2 phase mixture.

        Process_Input_Port feed;//Process inlet.
        Process_Output_Port vapour,liquid;//Process outlets
        Signal_Output_Port level_sig, press_sig, temp_sig;
        //Signal "transmitters" .
        Energy_Input_Port heat;//Heat duty.

        void feed_in(Stream& str);//Connection functions.
        void heat_in(Stream& str);
        void liquid_out(Stream& str);
        void vapour_out(Stream& str);

```

```

void level_out(Stream& str);
void temp_out(Stream& str);
void press_out(Stream& str);

void setup();//Input-output connection
int dynamic_model();//Model
void attach_compset(Component_Set& cs){
    VL_Mix.incorp_compset(cs);
    VL_mix.incorp_Thermo(peng_rob);
    VL_mix.incorp_VLE(peng_rob);
};
//attach set of Components to VL_mix object
//can define functions inside class header
//if desired
void initialise();//set initial estimates
void ss_output();//prints out steady-state
//solution.
Flash(int n, int no_comps, double vol,
       double diam);//Constructor.
//Creates equation structure and connects
//variables to port interfaces etc.
~Flash(){}; //Null destructor.
}; //End class declaration

int Flash::dynamic_model(){ //Start model

    int i;
    double CpL, CpV, hF, hL, hV, xsum, ysum, rhoL, rhoV;

    //Start physical properties.
    CpL = VL_mix.CpL(T(), P());
    CpV = VL_mix.CpV(T(), P());

    hF = VL_mix.H_mix(Tin(), Pin());
    hL = VL_mix.H_liq(T(), P());
    hV = VL_mix.H_vap(T(), P());

    VL_mix.calc_Ki(K, T(), P());
    rhoL = VL_mix.rhoL_molar(T(), P());
    rhoV = VL_mix.rhoV_molar(T(), P());
    //End physical properties.

    //Start equation evaluation.
    for(i=1; i<=nc; i++){
        de(i) = dNdt() - F()*z(i)() +
                L()*x(i)() + V()*y(i)(); //Equation 4.12
    }
}

```

```

de(nc+1) = (NL()*CpL + NV()*CpV)*dTdt() -
           Q() - F()*hF + L()*hL + V()*hV; //Equation 4.13

xsum = 0.0;
ysum = 0.0;

for(i=1;i<=nc;i++){
    eqbm(i) = y(i)() - K(i)*x(i)(); //Equation 4.14
    xsum = xsum + x(i)();
    ysum = ysum + y(i)();
}

mfs(1) = xsum - 1.0; //Equation 4.15
mfs(2) = ysum - 1.0; //Equation 4.16

for(i=1;i<=nc;i++){
    cmb(i) = N(i)() - Nv(i)() - Nl(i)(); //Equation 4.17
    lmb(i) = Nl(i)() - x(i)()*NL(); //Equation 4.18
    vmb(i) = Nv(i)() - y(i)()*NV(); //Equation 4.19
}

vb(1) = Vol - VL() - VV(); //Equation 4.20
vb(2) = VL() - NL()/rhoL; //Equation 4.21
vb(3) = VV() - NV()/rhoV; //Equation 4.22
vb(4) = h() - VL()/Area; //Equation 4.23

//End equation evaluation.

return OK;
} //end model.

```

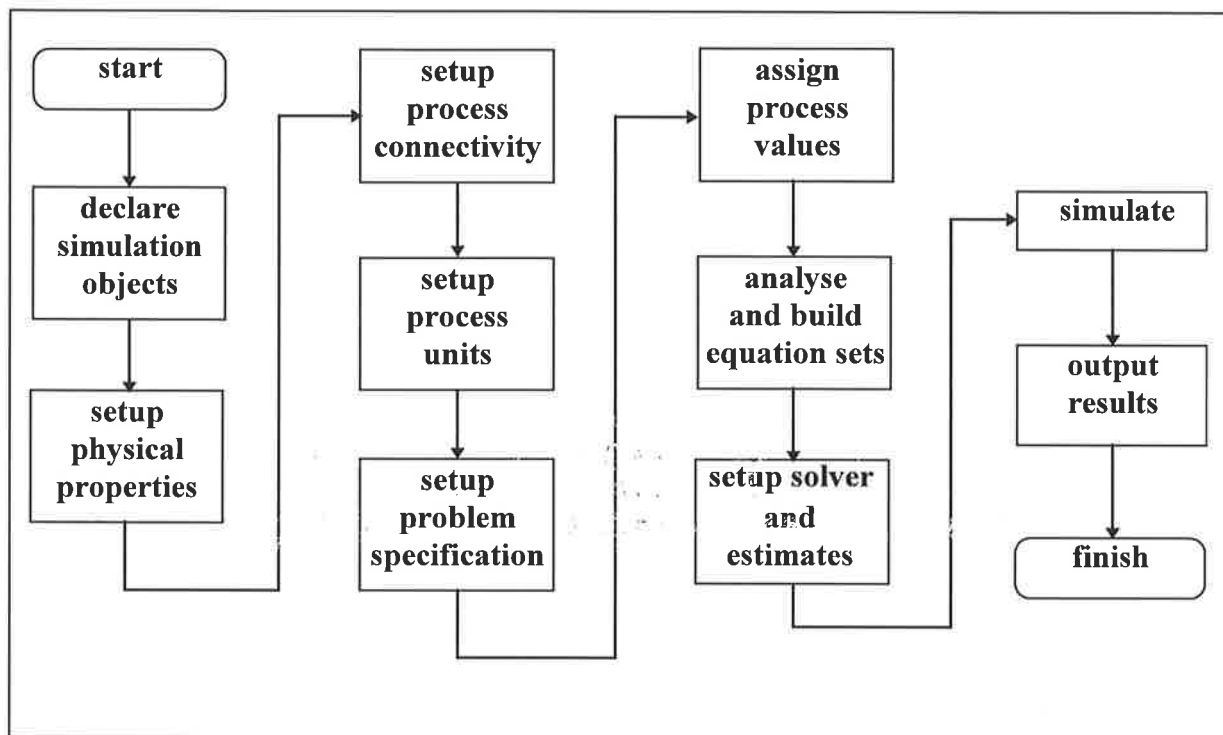
### 4.3 Simulation

Although this thesis is not intended to be a user's manual, a small example that performs a steady-state simulation of a **Flash** object serves as a useful introduction to simulation with the M.O.P.S. C++ classes.

#### 4.3.1 Instruction Sequence

A specific sequence of C++ instructions must be supplied in order to generate simulation code that will compile and run correctly. The objects and member functions of the M.O.P.S. C++ classes should not be manipulated in an *ad hoc* manner. The sequence of instructions for

describing a problem with the simulation classes is simple and logical. A flowchart of the steps required is presented in Figure 4.3. The flowchart is applicable to steady-state and dynamic simulation. If a steady-state simulation is employed to initialise a dynamic simulation, two instruction sequences are required. The steady-state sequence would be exactly as in Figure 4.3 but the dynamic sequence that follows would commence at the “setup problem specification” block. An example of a combined steady-state and dynamic simulation file is provided in Appendix D.



**Figure 4.3 Simulation instruction sequence.**

The “setup problem specification” and “assign process values” steps may be mixed together because it is more convenient in complex processes to assign specifications and values on a unit-by-unit basis.

#### 4.3.2 Steady-state example

A three-component steady-state flash example is demonstrated using the **Flash** class described earlier. The components are arbitrarily selected to be ethane, propylene and heptane. The flash conditions are 400.0 Kelvin and 5.0 bar absolute. The feed conditions are 450.0 Kelvin and 8.0 bar absolute. The reference conditions for the enthalpy calculations are 1.0 bar

absolute pressure and 298.15 Kelvin. Physical properties are calculated with the Peng-Robinson equation of state with binary interaction parameters set to zero.

The simulation input file for the example is presented below. The steps illustrated in Figure 4.3 are clearly indicated in the code. The file must be compiled and linked as part of the rest of the simulator structure.

```
#include " simstuff.hpp" /" simstuff.hpp" contains
//the general header files for various classes for
//flowsheet simulation, such as the Flowsheet and
//Process_Stream classes
#include <fstream.h>//include standard C++ header
//for file stream types
#include " flash.hpp" //include Flash class header file

void main(void){//start C++ program
    int i;// Counter.
    //DECLARE SIMULATION OBJECTS
    Flowsheet flwsht(999);//Declare a Flowsheet object.
    Flash flsh(1,3,10.0,1.5);//index no. 1,
    //3 components, 10 m^3 volume, 1.5 m diameter

    Process_Stream feed_str,vapour_str,liquid_str;
    Energy_Stream heat_str;

    Component ethane(" C2H6" ),propylene(" C3H8" );
    Component heptane(" NC7H16" );

    Component_Set comp_set(3);//Has 3 components.

    ifstream hcs("hydrocar.dat");//C++ ifstream object
    //containing hydrocarbon property data
    //FINISHED DECLARING SIMULATION OBJECTS

    //SETUP PHYSICAL PROPERTIES
    comp_set.incorp_comp(ethane,1);
    comp_set.incorp_comp(propylene,2);
    comp_set.incorp_comp(heptane,3);

    comp_set.set_datafile(hcs);
    comp_set.get_properties();

    flsh.attach_compset(comp_set);
    //FINISHED SETTING UP PHYSICAL PROPERTIES
```

```

//START PROCESS LAYOUT/CONNECTIVITY
flsh.feed_in(feed_str);
flsh.vapour_out(vapour_str);
flsh.liquid_out(liquid_str);
flsh.heat_in(heat_str);

flwsht.set_sys(1); //Only has one unit....
flwsht.incorp_sys(flsh,1); //...which is the flash drum.
//FINISHED PROCESS LAYOUT/CONNECTIVITY

//SETUP PROCESS UNITS
flwsht.setup(); //Set up the units in the flowsheet.
//FINISHED PROCESS UNIT SETUP

//START PROBLEM SPECIFICATIONS
flsh.Tin.constant(); //Feed and process conditions
flsh.Pin.constant(); //are specified parameters.
for(i=1;i<=3;i++) flsh.z(i).constant();
flsh.F.constant();

flsh.T.constant();
flsh.P.constant();
flsh.h.constant();
//FINISHED PROBLEM SPECIFICATIONS

//ASSIGN PROCESS VALUES
flsh.z(1) = 0.2; //Ethane feed mole fraction.
flsh.z(2) = 0.3; //Propylene feed mole fraction.
flsh.z(3) = 0.5; //Heptane feed mole fraction.
flsh.F = 0.05; //kmol/s. Equivalent to
//about 3.4 kg/s feed.

flsh.Tin = 450.0; //Feed temperature.
flsh.Pin = 8.0E5; //Feed pressure.
flsh.T = 400.0; //Flash temperature.
flsh.P = 5.0E5; //Flash pressure.
flsh.h = 2.5; //Liquid height in metres.
//FINISHED ASSIGNING PROCESS VALUES

//START ANALYSIS AND BUILDING OF EQUATION SETS
flwsht.ss_analyse(); //Analyse and build
flwsht.ss_build(); //equation structure.
//FINISHED ANALYSIS AND BUILDING OF EQUATION SETS

//SETUP SOLVER AND INITIAL ESTIMATES
flwsht.setup_solve(); //Send to solver.
flwsht.initialise(); //Sets up initial estimates
//for units in flowsheet

```



```

//SIMULATE THE PROBLEM
flwsht.solve_NEWT(); //Solve flowsheet.

//OUTPUT RESULTS
flsh.ss_output(); //Print out the answer.

//At this point the flowsheet's equation
//structure could be reset and re-analysed for
//dynamic simulation. Valves would be required
//on the vapour and liquid streams with specified
//pressure drop to create a full dynamic simulation.
//These would normally be solved for valve coefficients
//as part of the steady-state simulation to provide
//a consistent initialisation.

} //End of C++ program

```

**Process\_Stream**, **Energy\_Stream** and **Flowsheet** objects are employed for completeness although they are not actually necessary in this example because the **Flash** class owns all of the relevant **Variables**. In general, single-unit steady-state and dynamic simulations may be run with only an object of the class and correct problem specifications. Stand-alone numerical models without any connectivity may also be created and solved.

The results for the simulation are in Table 4.1. Compositions are in mole fractions.

|                            | <i>Feed</i> | <i>Vapour</i> | <i>Liquid</i> |
|----------------------------|-------------|---------------|---------------|
| <b>Temperature (K)</b>     | 450.0       | 400.0         | 400.0         |
| <b>Pressure (bar)</b>      | 8.0         | 5.0           | 5.0           |
| <b>ethane</b>              | 0.2000      | 0.2194        | 0.0133        |
| <b>propylene</b>           | 0.3000      | 0.3268        | 0.0416        |
| <b>heptane</b>             | 0.5000      | 0.4538        | 0.9451        |
| <b>Total Flow (kmol/s)</b> | 0.05        | 0.0453        | 0.0047        |
| <b>Cooling Duty (kW)</b>   | 724         |               |               |

**Table 4.1: Composition and duty for vapour-liquid flash calculation.**

#### 4.4 Summary

Several model decomposition techniques were presented in this chapter. Medium and Machine Decomposition divides the model into the physical object (machine) and vessel mixture (medium). Primitive Behaviour Decomposition creates composite models from sub-objects that are in turn based on modelling principles and assumptions (e.g. conservation of mass). Mathematical Decomposition is based on organising the equations in a model into sets according to the equations' purpose in the model (e.g. mass balance, equilibrium, steady-state or dynamic *etc.*).

Modelling examples were provided to demonstrate various characteristics of the simulator data structure and modelling approaches. The basic data structure was demonstrated with a **Mixing\_Tank** class. Aggregation of model characteristics with connected objects and multiple inheritance of characteristics were contrasted using a **Ratio\_Controller** class. Physical property modelling was demonstrated with a **Flash** class.

Finally, simulation with the M.O.P.S. data structure was introduced with a small steady-state example.

# CHAPTER 5

## Major Test Problems

The simulator has been tested with three major plant process models, viz. the four-flash Cavett problem (Cavett 1963), the Tennessee Eastman Process (Downs and Vogel 1993) and a recombinant fermentation model, developed from various model elements in the literature.

### 5.1 Cavett Problem

The well-known Cavett problem contains four flash vessels with three recycle streams, as illustrated below in Figure 5.1. The mathematical model for the flash calculations is different from the model described in the previous example.

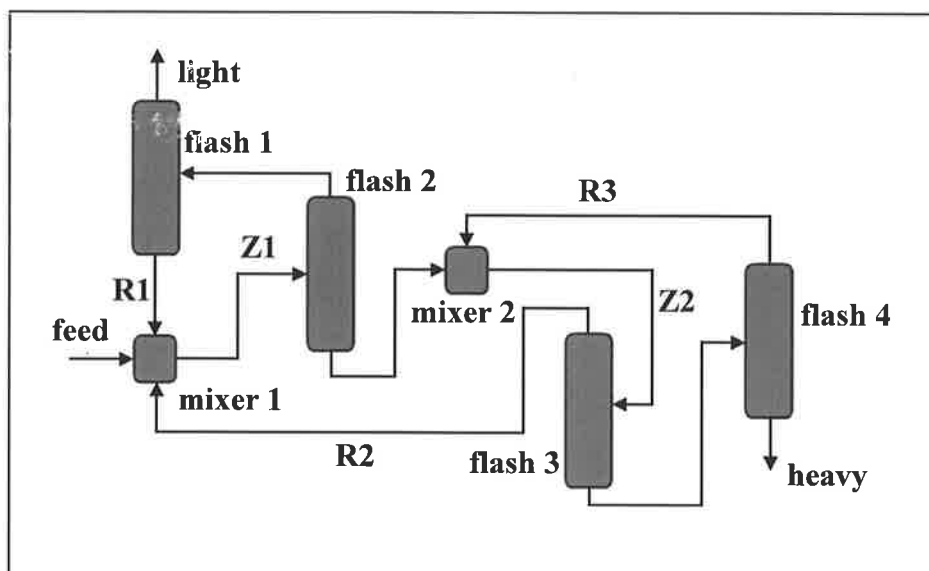


Figure 5.1: Cavett Process.

The process is basically a stripping operation with four ideal stages. There are sixteen components in the process: nitrogen, carbon monoxide, hydrogen sulphide, methane, ethane, propane, *n*- and *i*-butane, *n*- and *i*-pentane, hexane, heptane, octane, nonane, decane and undecane.

The problem was originally proposed as a small but stringent test for sequential-modular simulators. While trivial compared to the Tennessee Eastman test problem (see section 5.2), the Cavett process was appropriate as a steady-state development problem for this project. The Cavett process was employed to refine the **Equation\_of\_State** physical property classes (specifically the **Peng\_Robinson** class) and to develop interchangeable solution methods within the **Convergence\_Block** class.

It was not an objective to thoroughly investigate the convergence behaviour of the Cavett problem with different tear sets as many researchers have already accomplished this (Lau 1992 and Chen and Stadtherr 1985 are two good examples). The tear set chosen was streams Z1 and Z2, given by a synthetic tearing algorithm (Roach *et al.* 1996). The principal objectives were:

- to investigate sequential-modular initialisation and assistance of equation-oriented simulation.
- to demonstrate the variety of simulation techniques available with the simulator data structure.

A secondary objective was to investigate sequential- and parallel-modular simulation of the Cavett problem while employing an equation-oriented unit solution. Equation-oriented unit solution was chosen to facilitate the specification of design and rating simulations.

The rating calculations were based on a mass balance around the process. The design calculations were based on a mass balance with specification of i-butane recovery into the vapour stream from the second flash unit (see Figure 5.1). The specification was 50% recovery of the i-butane in stream Z1. Energy balances were not performed. Physical property calculation was based on the Peng-Robinson equation of state with binary interaction parameters set to zero.

The feed and product stream compositions for the rating and design simulations are summarised in Table 5.1.

| Component                                | Rating                 |                |                | Design         |                |
|--|------------------------|----------------|----------------|----------------|----------------|
|  | Feed (kmol/s)          | Light (kmol/s) | Heavy (kmol/s) | Light (kmol/s) | Heavy (kmol/s) |
|  | $T_{flash2} = 365.0$ K |                |                |                |                |
| N <sub>2</sub>                           | 0.04523                | 0.04348        | 0.00175        | 0.04327        | 0.00196        |
| CO                                       | 0.62697                | 0.62694        | 0.00003        | 0.62692        | 0.00005        |
| H <sub>2</sub> S                         | 0.04285                | 0.03562        | 0.00723        | 0.03562        | 0.00723        |
| CH <sub>4</sub>                          | 0.37822                | 0.37792        | 0.00030        | 0.37783        | 0.00039        |
| C <sub>2</sub> H <sub>6</sub>            | 0.30246                | 0.28303        | 0.01943        | 0.28155        | 0.02091        |
| C <sub>3</sub> H <sub>8</sub>            | 0.28927                | 0.15351        | 0.13577        | 0.16278        | 0.12649        |
| <i>i</i> -C <sub>4</sub> H <sub>10</sub> | 0.07628                | 0.01832        | 0.05796        | 0.02044        | 0.05584        |
| <i>n</i> -C <sub>4</sub> H <sub>10</sub> | 0.19443                | 0.03357        | 0.16086        | 0.03711        | 0.15732        |
| <i>i</i> -C <sub>5</sub> H <sub>12</sub> | 0.09980                | 0.00685        | 0.09295        | 0.00682        | 0.09298        |
| <i>n</i> -C <sub>5</sub> H <sub>12</sub> | 0.14266                | 0.00725        | 0.13501        | 0.00702        | 0.13524        |
| C <sub>6</sub> H <sub>14</sub>           | 0.22282                | 0.00301        | 0.21981        | 0.00257        | 0.22025        |
| C <sub>7</sub> H <sub>16</sub>           | 0.32913                | 0.00104        | 0.32809        | 0.00087        | 0.32826        |
| C <sub>8</sub> H <sub>18</sub>           | 0.23289                | 0.00016        | 0.23273        | 0.00014        | 0.23275        |
| C <sub>9</sub> H <sub>20</sub>           | 0.21073                | 0.00003        | 0.21070        | 0.00003        | 0.21070        |
| C <sub>10</sub> H <sub>22</sub>          | 0.10501                | 0.00000        | 0.10501        | 0.00000        | 0.10501        |
| C <sub>11</sub> H <sub>24</sub>          | 0.15335                | 0.00000        | 0.15335        | 0.00000        | 0.15335        |
| Temperature (K)                          | 322                    | 311            | 303            | 311            | 303            |
| Pressure (kPa abs.)                      | 439                    | 5620           | 191            | 5620           | 191            |

**Table 5.1: Cavett feed and product stream compositions.**

The rating specifications for each flash vessel are in Table 5.2. The design simulation calculated the temperature of the second flash unit to be 365.0 K.

| Unit    | Temperature<br>(K) | Pressure<br>(kPa abs.) |
|---------|--------------------|------------------------|
| flash 1 | 311                | 5620                   |
| flash 2 | 322                | 1960                   |
| flash 3 | 309                | 439                    |
| flash 4 | 303                | 191                    |

**Table 5.2: Cavett flash specifications.**

The number of iterations and time in seconds to convergence are presented in Table 5.3 for rating simulations and in Table 5.4 for design simulations. The results are presented as **iterations (time)** in each cell. Each simulation was run from three initial tear estimates of  $x_0$ ,  $0.1x_0$  and  $10x_0$ . The estimate  $x_0$  corresponds to a tear stream estimate of equimolar composition and a flow of 0.5 kmol/s. Parallel-modular and equation-oriented simulations were initialised with one sequential-modular iteration around the flowsheet. Each **Flash** object contained 16 equations and each **Mixer** object contained 8 equations, to yield a total of 160 equations in the equation-oriented system. The unit models for each simulation run were identical. Jacobian matrices were calculated with central differences. Each tear stream contained 16 variables. The simulations were performed on an IBM 486 DX33 with 16MB RAM under the OS/2 operating system.

| Initial estimate | Simulation Method  |             |                           |         |             |                                |         |           |
|------------------|--------------------|-------------|---------------------------|---------|-------------|--------------------------------|---------|-----------|
|                  | Sequential-modular |             | Parallel-modular          |         |             | Equation-oriented              |         |           |
|                  | Dir Subst          | Wegs        | Newt                      | Broy    | Marq        | Newt                           | Broy    | Marq      |
| $0.1x_0$         | 49 (130)           | 50<br>(140) | 6 (127)<br><i>seq. 2*</i> | 13 (47) | 16 (331)    | 7 (188)                        | 12 (53) | 3 (144)   |
| $x_0$            | 51 (134)           | 50<br>(135) | 6 (128)<br><i>seq. 2*</i> | 13 (44) | <i>fail</i> | 5 (136)                        | 23 (66) | 4 (185)   |
| $10x_0$          | 74 (296)           | 73          | 6 (160)<br><i>seq. 2*</i> | 13 (76) | <i>fail</i> | 29 (1285)<br><i>seq. 2, 4*</i> | 35 (84) | 12 (1500) |

**Table 5.3: Iterations and solution time to convergence for the Cavett rating problem.**

\* *seq. n* indicates where an equation-oriented method encountered convergence difficulties on iteration *n*. At this point the **Convergence\_Block** object switched to a sequential-modular simulation for one iteration and then switched back to the equation-oriented method.

| Initial estimate | Simulation Method  |             |                            |          |             |                            |                            |           |
|------------------|--------------------|-------------|----------------------------|----------|-------------|----------------------------|----------------------------|-----------|
|                  | Sequential-modular |             | Parallel-modular           |          |             | Equation-oriented          |                            |           |
|                  | Dir Subst          | Wegs        | Newt                       | Broy     | Marq        | Newt                       | Broy                       | Marq      |
| $0.1x_0$         | 49 (234)           | 48<br>(179) | 6 (756)<br><i>seq. 2*</i>  | 15 (161) | <i>fail</i> | 7 (182)                    | 14 (57)                    | 3 (194)   |
| $x_0$            | 53 (208)           | 53<br>(210) | <i>fail</i><br>( $>10^4$ ) | 15 (162) | <i>fail</i> | 25 (716)                   | 86 (142)                   | 4 (189)   |
| $10x_0$          | 76 (497)           | 80<br>(455) | 7 (1148)<br><i>seq. 2*</i> | 16 (185) | <i>fail</i> | 44 (728)<br><i>seq. 2*</i> | 36 (202)<br><i>seq. 2*</i> | 22 (1809) |

**Table 5.4: Iterations and solution time to convergence for the Cavett design problem.**

\* *seq. n* indicates where an equation-oriented method encountered convergence difficulties on iteration *n*. At this point the **Convergence\_Block** object switched to a sequential-modular simulation for one iteration and then switched back to the equation-oriented method.

The majority of equation-oriented simulations did not require sequential-modular assistance for convergence. However, for this problem it was considerably easier to commence an equation-oriented simulation with a sequential-modular iteration because only 32 tear stream variables had to be estimated out of the 160 variables in the complete flowsheet, because the

flash vessels contain their own initialisation routines. Marquardt's method was the most efficient numerically and Broyden's method the fastest to solve in the equation-oriented simulations.

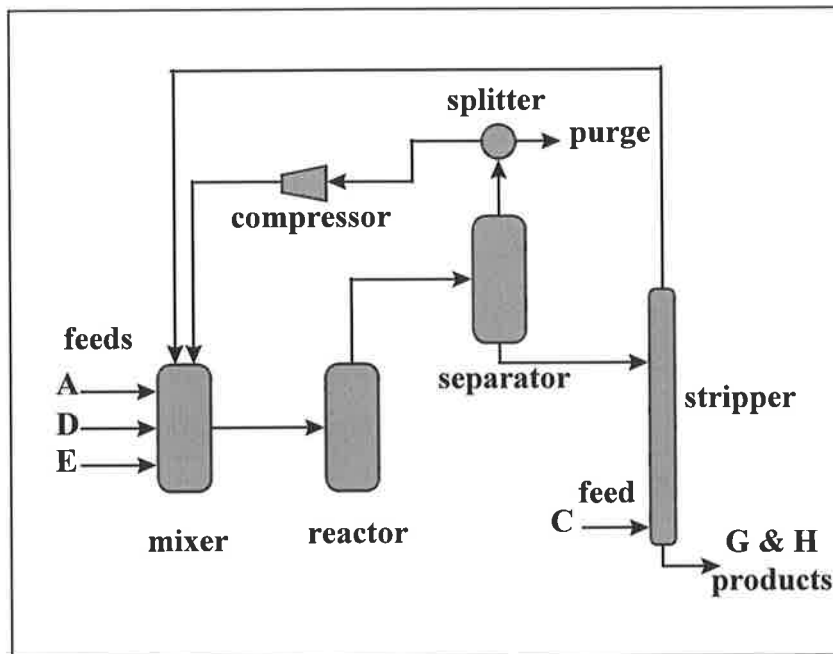
The iterations for sequential- and parallel-modular simulations are similar for the successful design and rating calculations. This result is reasonable; there was only a single design specification, hence the iterations to convergence should not be affected greatly. A similar result could probably be expected with a traditional "design convergence loop" around a **Flash** unit. The main advantage of equation-oriented unit solution is that design and rating calculations are considerably easier to manage at the flowsheet executive level. The design calculation times were longer because generally a design flash calculation required 30 - 50% more iterations to converge compared to a similar rating calculation. Marquardt's method performed extremely poorly on parallel-modular simulations due to problems with calculation of the feed:vapour ratio in the third flash vessel. Newton's method was terminated on one of the parallel-modular simulations because 5 iterations required over 10000 seconds to complete with little convergence progress.

None of the numerical methods or simulation techniques demonstrated themselves to be superior. While testing one process does not provide conclusions about steady-state simulation in general, it is clear that there are potential advantages in having a variety of interchangeable numerical and flowsheet solution methods available. In particular, sequential-modular initialisation of equation-oriented flowsheet solution is a very useful feature. Equation-oriented unit solution within a sequential-modular simulation provides the advantage of simple design specification (provided the unit models are written correctly) with the lower resource requirements of sequential-modular simulation.

## **5.2 Tennessee Eastman Process**

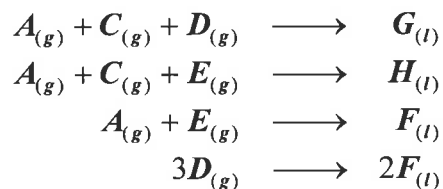
The Tennessee Eastman Process was proposed several years ago as a dynamic process-control test problem. The basic process layout is drawn in Figure 5.2. The layout is slightly different from that provided in the original paper.





**Figure 5.2: Tennessee Eastman Process.**

There are eight chemical species in the process: A,B,C,D,E,F,G and H. The process produces two products (G and H) and a by-product (F) from A,C,D and E from the following exothermic reactions (B is an inert component):



Three vapour-phase reactants and two recycle streams are mixed in a vapour-phase mixing section. The reactor is two-phase with a vapour product. The reactor product enters a separation vessel. The vapour stream of the separator contains primarily unreacted feed components. A small fraction of the stream is purged and the remainder is compressed and recycled. The separator liquid enters a stripping column where the major products are removed in the bottoms stream and the overhead stream is recycled. The plant contains dead-time and discontinuities from the concentration analysers on the reactor feed stream, purge stream and product stream. The reactor feed-stream and purge-stream analysers sample every 0.1 hours and have a 0.1 hour dead-time, the product analyser samples every 0.25 hours and has a 0.25 hour dead-time. The plant dynamics are slightly stiff.

The original problem was distributed as a complex piece of Fortran code. Mathematical models (such as reaction kinetics) were not provided explicitly. The code contained various arrays through which the major process variables could be accessed. Incorporation of Fortran code into C++ code is feasible. If the code was incorporated into this simulator project, it would produce one **System**-type object with very complex internals. However, by attaching **Variable** and **Derivative** objects to the relevant arrays in the Fortran model and defining a **Vector** of **Equation** objects, the code could become the `dynamic_model()` of a large **Tennessee\_Eastman** class, with appropriate **Signal\_Input\_Ports** and **Signal\_Output\_Ports** for controller connections.

Incorporation of the Fortran code does not really test the modelling capabilities of the data structure. With a view to modelling the unit operations separately, the Fortran code was reverse-engineered to determine the reaction kinetics and basic unit operation principles. The deconstruction was time-consuming but successful. The separator was modelled as a simple flash drum with three incondensable components and the stripper was modelled with a temperature-dependent vapour recovery as in the original code. The heat duty of the stripper published in the original Tennessee Eastman paper exceeds that calculated by the Fortran code by factor of 10. The reactor product condenser of the original process was incorporated into the separator drum. The dead-time and discontinuities in the analysers are incorporated. While the Fortran code contained physical property methods based on temperature-dependent specific heat, the basic pure component data provided in the original paper was sufficient to permit the use of the ideal physical property classes described in earlier chapters. The complete models for the major unit operations are provided in Appendix C.

### 5.2.1 Control Systems

Many papers have been published on various control strategies for the Tennessee Eastman problem. Nonlinear Model Predictive Control (NMPC) was investigated by Ricker and Lee (1995a). The process model employed was theoretically based with adjustable model parameters (Ricker and Lee 1995b). A NMPC scheme was outside the scope of the project. NMPC could be implemented with the existing class structure however; it is a matter of defining the appropriate controller algorithms in **System**-based classes. Various SISO formulations are described in Lyman and Georgakis (1995), Ye and McAvoy (1995), Banerjee and Arkun (1995) and McAvoy and Ye (1994). A very simple and effective SISO scheme is

proposed by Luyben (1996). The production rate out of the plant is controlled by the valve on the stripper bottoms, giving almost instantaneous production rate control. The liquid level control loops operate opposite to the direction of flow.

Initially some of the cascaded SISO schemes suggested by McAvoy and Ye (1994) were simulated but the reactor pressure control was poor and the results of McAvoy and Ye could not be duplicated. The reverse-engineered plant model of this project is slightly different from the original Fortran model and this may account for the control behaviour. The reactor pressure was controlled by manipulating the flow control setpoint of component A (stream 1) to the mixer. Normally, this has a positive gain: a higher concentration of component A increases the reaction rate and the pressure decreases as a result of product condensation. However, the gain of the reactor pressure relative to component A changes sign when the partial pressure of A is high in the reactor (Ricker and Lee 1995a). At high concentrations of A, the reaction rate decreases due to the low concentrations of the other reactants and consequently the pressure increases. A scheme of Banerjee and Arkun (1995) with reactor pressure controlled by manipulating the reactor cooling water flow was also tested. This provided exceptionally tight control of reactor pressure.

Based on the excellent results obtained with the Banerjee and Arkun scheme it was decided to investigate whether a pressure-temperature reactor control system could be combined with the Luyben production-rate scheme. The complete Luyben scheme controls the production rate with liquid flow valve from the stripper. This provides the fastest control on production rate. The separator drum liquid flow valve then controls the level in the stripper. Manipulating the cooling duty of the separator controls the separator level. Reactor pressure is controlled with the C feed stream. Reactor level is controlled by manipulating the D and E feed valves.

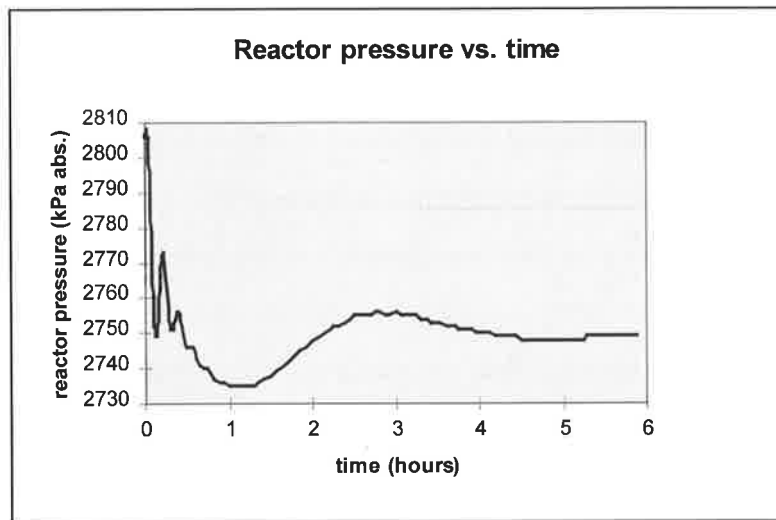
In the combined scheme, the reactor pressure was controlled by manipulating the reactor temperature. The reactor liquid level was controlled with the E feed stream. The D feed stream flow was regulated by a composition controller on the product stream to control the mass ratio of the products G and H. The purge composition of component B was controlled by the purge flow valve.

Initially, the reactor pressure controller manipulated the reactor cooling water valve directly. This system was unstable unless tuned for an unacceptably slow response. The addition of a cascaded PI reactor temperature controller made the pressure more rapidly controllable, but produced an oscillatory response. The cascaded controller was not required in the Banerjee and Arkun scheme. It was also found that the level in the separator could not be controlled by manipulating the separator temperature. To reduce reactor pressure, the reactor temperature is increased. This increases the rate of production of heavy components F,G and H, which in turn increases the level in the separator. If the separator level is controlled by temperature, the level controller will decrease the separator cooling to vaporise the heavier components and reduce the level. The heavier components then traverse the recycle loop and dilute the pure reactant concentrations, thereby reducing the reaction rate. In addition, the extra heavy components in the separator vapour increase the amount of gas in the volume of the recycle system, which also increases pressure. This control system contains a significant amount of positive feedback.

Stabilising effects are provided by the reactor level controller, product ratio controller and the stripper product flowrate. With more heavy components in the recycle, the reactor level will increase. Therefore the level controller will decrease the E feed and the ratio controller will eventually reduce the D feed. The reduced reactant concentrations would reduce the reaction rate further. The stripper product flowrate would then deplete the separator level. However, in practice, the system saturated or reached shutdown limits before it could stabilise, typically in five hours or less. This indicates that the effect of temperature on reaction rate greatly exceeds the combined effects of the reduced reactant feeds and concentrations. The separator temperature continued to increase to the point where the liquid feed to the stripper was sufficiently hot to close the stripper reboiler steam valve, thus decoupling the stripper temperature control loop. The increasing stripper temperature then forced more heavy products back up into the recycle loop, increasing the pressure further. Depending on controller tuning, the reactor pressure reached the shutdown limit or the separator would overflow without stabilising. Pure proportional control of the plant was also unstable with this scheme.

Examination of the unstable responses revealed a possible solution. The production rate in the reactor should obviously balance the flowrate of products from the stripper base. The rate of change of level in the separator indicates the balance between the reactor production rate and stripper flowrate. If they are out of balance the separator will either run dry or overflow. The control objectives for the problem state that reactor pressure should be independently controllable. If the reactor pressure is controlled by manipulating reactor temperature, two variables for directly manipulating the reaction rate are unavailable. The remaining possibilities for reaction rate control are the various feed streams. Four streams are available (A feed, C feed, D feed and E feed). Only components A and C affect all reactions. The A feed stream was considered too small to provide effective control and was instead tied to a purge composition controller to maintain the component A concentration in the reaction loop. The C feed stream is the largest feed stream to the process and should provide the best control, provided that there are no adverse effects on other process variables. The separator level was therefore controlled with the C feed stream. The A composition control loop ensured that only the concentration of component C affected the reaction rate. The separator temperature was controlled by manipulating the separator cooling water flowrate. This also helped to reduce temperature disturbances into the stripper. The stripper temperature was controlled by regulating the steam flow into the reboiler.

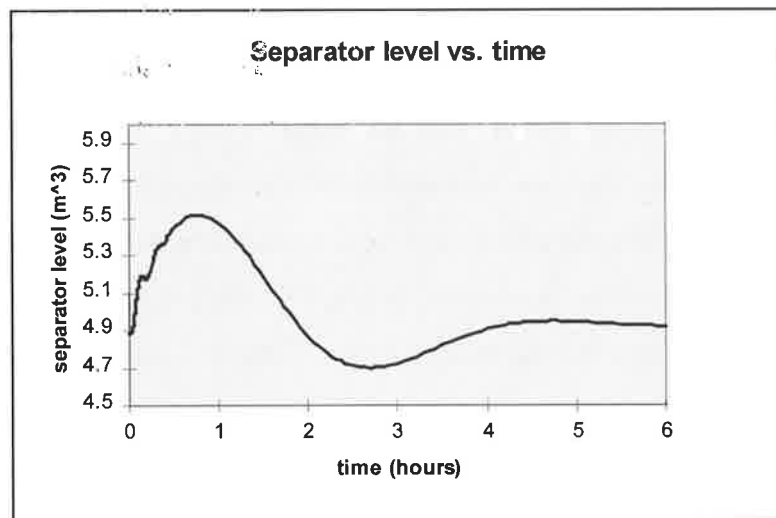
Simulations of this control system gave, at best, erratic pressure control of the reactor. The adjustments to the C feed rate by the separator level control seriously affected the behaviour of the reactor pressure control loop. Although some pressure effects were expected, it was hoped that the pressure/temperature cascade would be sufficiently robust and fast enough to override the disturbances. This proved not to be the case. Simulation results for the reactor pressure setpoint change are provided in Figure 5.3.



**Figure 5.3: Reactor pressure response.**

**Reactor pressure setpoint change from 2806 to 2746 kPa absolute.**

The corresponding separator level response is in Figure 5.4.



**Figure 5.4: Separator level response.**

**Reactor pressure setpoint change from 2806 to 2746 kPa absolute.**

Obviously the effect of the C feed stream on the reactor pressure exceeds the capability of the pressure control loop. Following the setpoint change, the reactor temperature increases to decrease the reactor pressure. This increases the reaction rate and hence the flow of heavy products into the separator increases, which increases the separator level, in spite of the lower

pressure. Therefore the separator level controller reduces the C feed, which slows the reaction rate. Again the reactor pressure rises. As the separator level decreases and undershoots, the C feed is increased, which increases the reaction rate and the system settles down.

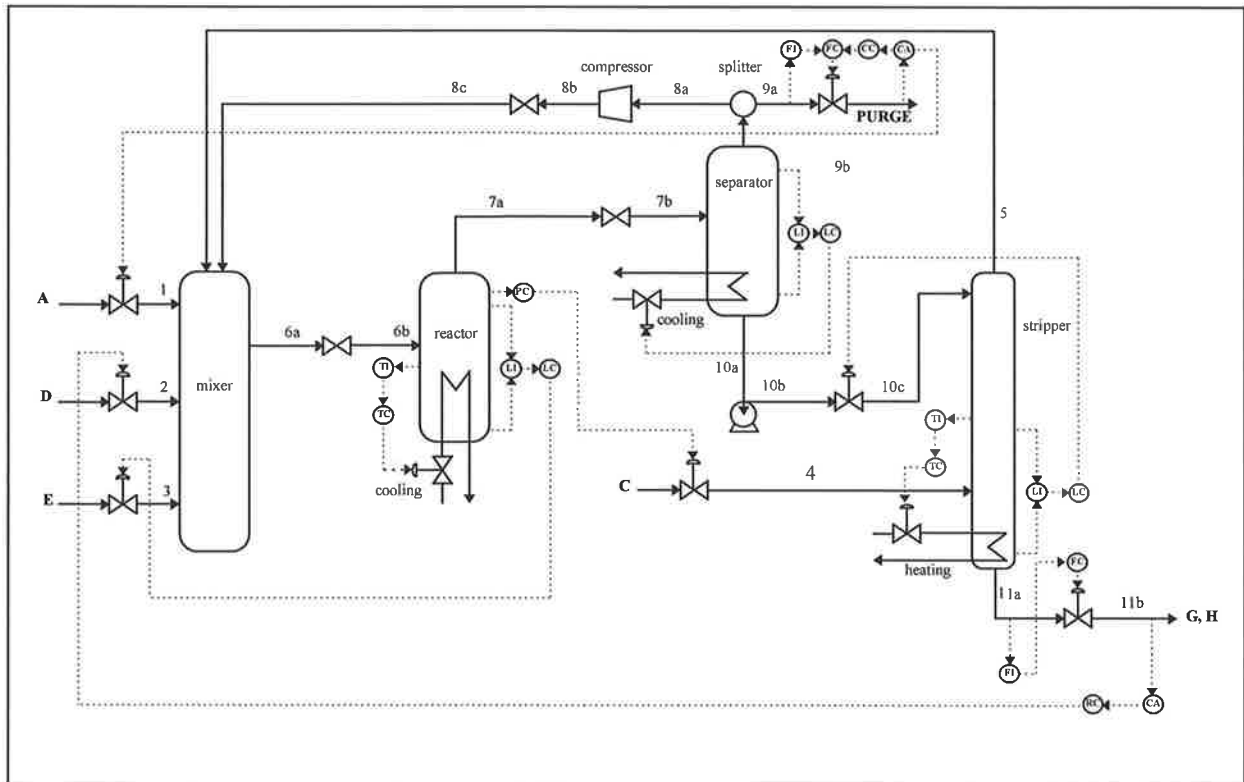
The pressure control loop could not be tuned more aggressively because the reactor temperature controller could not cope with faster setpoint changes. The spikes in the reactor pressure response in Figure 5.3 are due to the reactor temperature controller overshooting slightly. The response time of the temperature loop could possibly have been decreased with derivative action but the original problem statement declared that the process measurements in the Fortran code were noisy and so derivative action was not considered. An alternative, much slower pressure control loop was tested, but the effects of the C feed stream changes caused the reactor pressure to oscillate for many hours before stabilising. A slower separator level control loop assisted the reactor pressure control, but caused the separator to overflow. Further fine-tuning of the system could have provided improved performance but the basic system is strongly coupled and only marginally stable for separator level control. The scheme was abandoned with no further testing. Good control of product composition was maintained in all the tests.

The most successful control system simulated was a modification of that proposed by Luyben (1996). There are three major deviations from the original Luyben scheme:

1. The reactor level is controlled with only the E feed stream and the product composition is controlled by a ratio controller manipulating the D feed stream.
2. In the Luyben scheme, a separator temperature controller manipulates the reactor temperature controller setpoint. This loop was deleted.
3. Proportional-integral controllers are incorporated, whereas the Luyben scheme employs only proportional controllers.

The major control loops are not cascaded to flow control loops. While cascaded flow control loops are good control practice, they produced no difference to the final simulation results and were omitted in the interests of computational speed. The Luyben reactor pressure control loop is very effective. The reactor pressure is controlled by regulation of the C feed stream. The C feed stream simply adjusts the amount of gas circulating through the compression

recycle loop. Integral action is applied to vessel temperatures and to controlled variables with specific objectives in the problem statement. The product flow and composition, purge composition, separator, stripper and reactor temperature and reactor pressure controllers are proportional-integral. Vessel level controllers are proportional only. The final scheme is illustrated below in Figure 5.5.



**Figure 5.5: Modified Luyben Control Scheme for Tennessee Eastman Process.**

The original paper specifies four main process changes to evaluate performance:

1. **Production rate change:** Make a step change to the production rate control variables to reduce the production rate by 15% on a mass basis.
2. **Product mix change:** Make a step change to the product composition control variables to adjust the production rates of products G and H from 7038 kg/h each to 5630 kg/h G and 8446 kg/h H. This is a 50:50 to 40:60 change.
3. **Reactor operating pressure change:** Make a step change to the reactor operating pressure setpoint from 2705 kPa (gauge) to 2645 kPa (gauge).
4. **Purge gas composition change:** Make a step change so that the composition of component B in the purge changes from 13.82 mole % to 15.82 mole %.



These changes were simulated with the Luyben-based scheme described previously. The results are presented in the next section (5.2.2). The process conditions prior to the setpoint changes were determined by performing an equation-oriented, steady-state simulation with the dynamic plant models. (Time derivatives were set to zero). The steady-state simulation calculated process conditions, all holdups, cascade setpoints and controller bias signals. This ensured consistent initialisation of the dynamic system. The full dynamic plant model contained between 190 - 220 equations, depending on the control system being simulated. About 50 - 70 equations were dynamic and the remainder algebraic. Physical properties were not part of the flowsheet equation set. The initial estimate for the steady-state simulation was provided by the process conditions outlined in Downs and Vogel (1993). As stated earlier, the plant model differed slightly from the original Fortran model and about five iterations were required to converge the steady-state.

The dynamic simulations were solved with the variable-step, variable-order BDF integration method from the **Mathtool** class. The stiffness of the original problem is increased by the many different time constants of the control system. A complete definition file for the reactor pressure setpoint change simulation is provided in Appendix D. The code in the definition file is reasonably self-explanatory and follows a similar specification format to other simulators employing a text-based input. Each time the file is modified it must be recompiled for the new simulation to run. The analysis and construction steps for the equation sets indicate the number of variables and equations participating from each unit to assist problem specification.

### 5.2.2 Simulation Results

The time responses for the major process variables are presented on the following pages. Results for the original control scheme using the Fortran model are presented in Luyben (1996). The original Fortran model contained built-in plant noise. The absence of noise in the C++ model makes controlling the process easier. Controller tuning constants are provided in the flowsheet specification in Appendix D.

Referring to Figure 5.6, the production rate control is almost instantaneous. The problem specification states that variations in product flow of greater than  $\pm 5\%$  with a frequency between 8 - 16 h<sup>-1</sup> are harmful to downstream processes. The mole fraction of component G

in the product stream should not vary by more than  $\pm 5\%$  with a frequency of 6 - 10 h<sup>-1</sup>. The product flow exhibits no variation after it reaches the new setpoint. The product composition exhibits one oscillation between 2 - 6 hours but the variation is within the specification. The oscillation is caused by the reactor level controller.

The stripper temperature plot requires further explanation. The controller setpoint was 338.8 K. The decreased production rate raises the level in the separator. The separator level is controlled by regulating the separator cooling water flow. With a higher separator level, the level controller will increase the temperature to reduce the level. Although not shown on the plots, the separator temperature increased by about 10 degrees. The higher separator liquid temperature meant that the stripper steam was no longer required and the temperature controller output saturated at zero. After about six hours the stripper temperature stabilised. The simulator data structure permits connections between **Variable** objects to be broken and remade without affecting the equation structure. The **Controller** classes exploit this feature. The principle is very simple. The stripper steam valve's position **Variable** is connected to the controller's output **Variable**. The controller's **Variable** is thus the "driver" and is the **Variable** that the numerical method manipulates. At saturation, the controller finds the steam valve **Variable** by interrogating the appropriate **Signal\_Output\_Port**. Informally, the controller looks down the **Signal\_Stream** to get the **Variable** at the other end. The steam valve's **Variable** is then disconnected from the output **Variable** and frozen at the saturated value. The valve position is no longer a process input. The controller's output **Variable** remains in the solution set and its value changes with time as it did prior to the disconnection. In addition, at disconnection a controller switches off the integral action, resets the integrated error term to zero and flags a discontinuity for the integration to restart. If the stripper temperature had decreased below the controller setpoint, the controller would have reconnected the valve and resumed control action. The reconnection behaviour is the opposite to disconnection. The decoupling could possibly be prevented by incorporating the separator temperature control loop of the original Luyben scheme.

Figure 5.7 contains the responses to the product ratio setpoint change. The product ratio is changed and stable in about three hours. The product molar flowrate decreases because of the composition change. The stream is controlled based on the volume flow. The mass flowrates

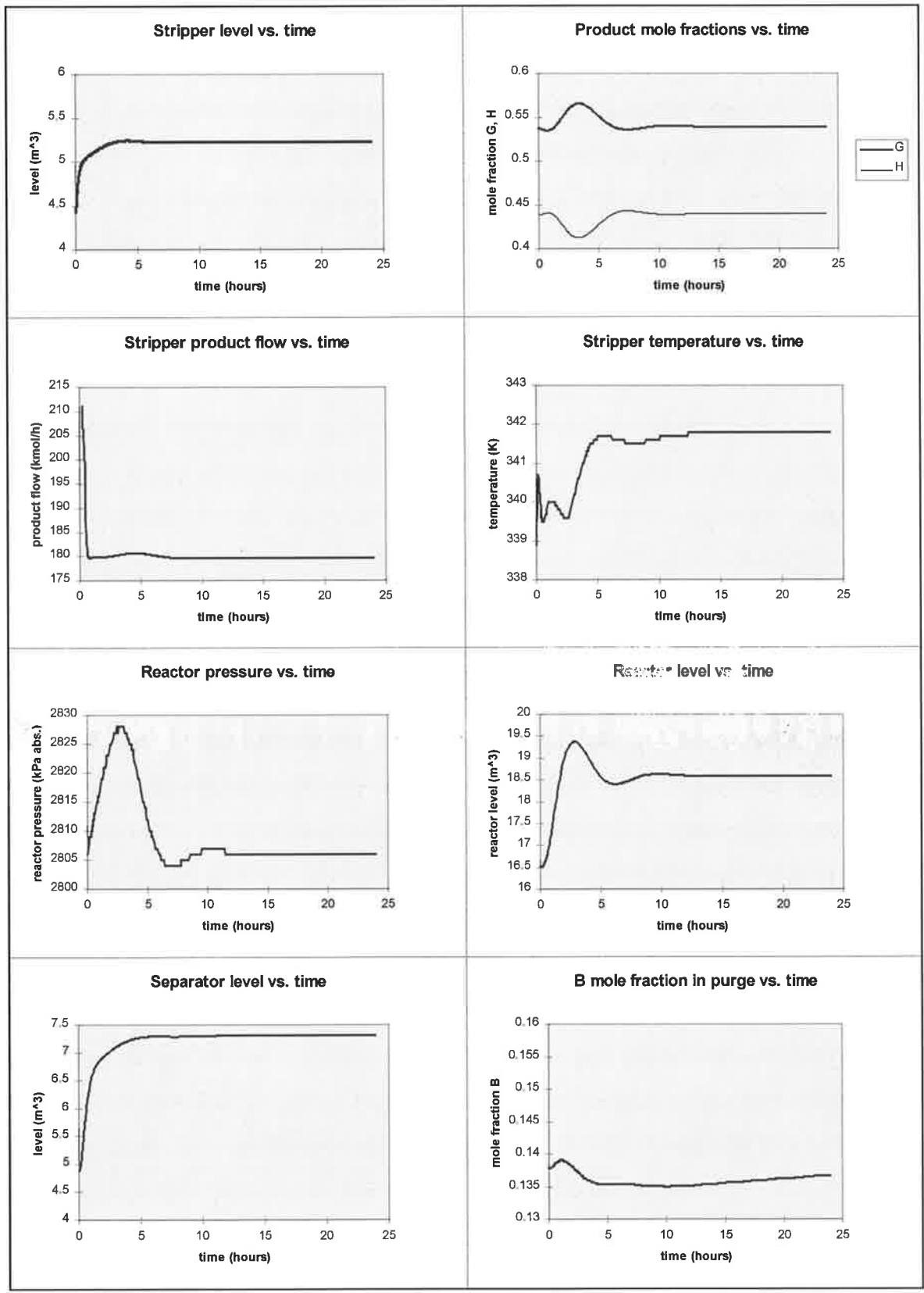
of the products are within one percent of the specification. The other process variables remain well-controlled.

The responses for the reactor setpoint change are in Figure 5.8. The reactor pressure responds rapidly, overshooting very slightly by about three kPa before stabilising at the new setpoint. In spite of the large change, the rest of the process remains relatively undisturbed.

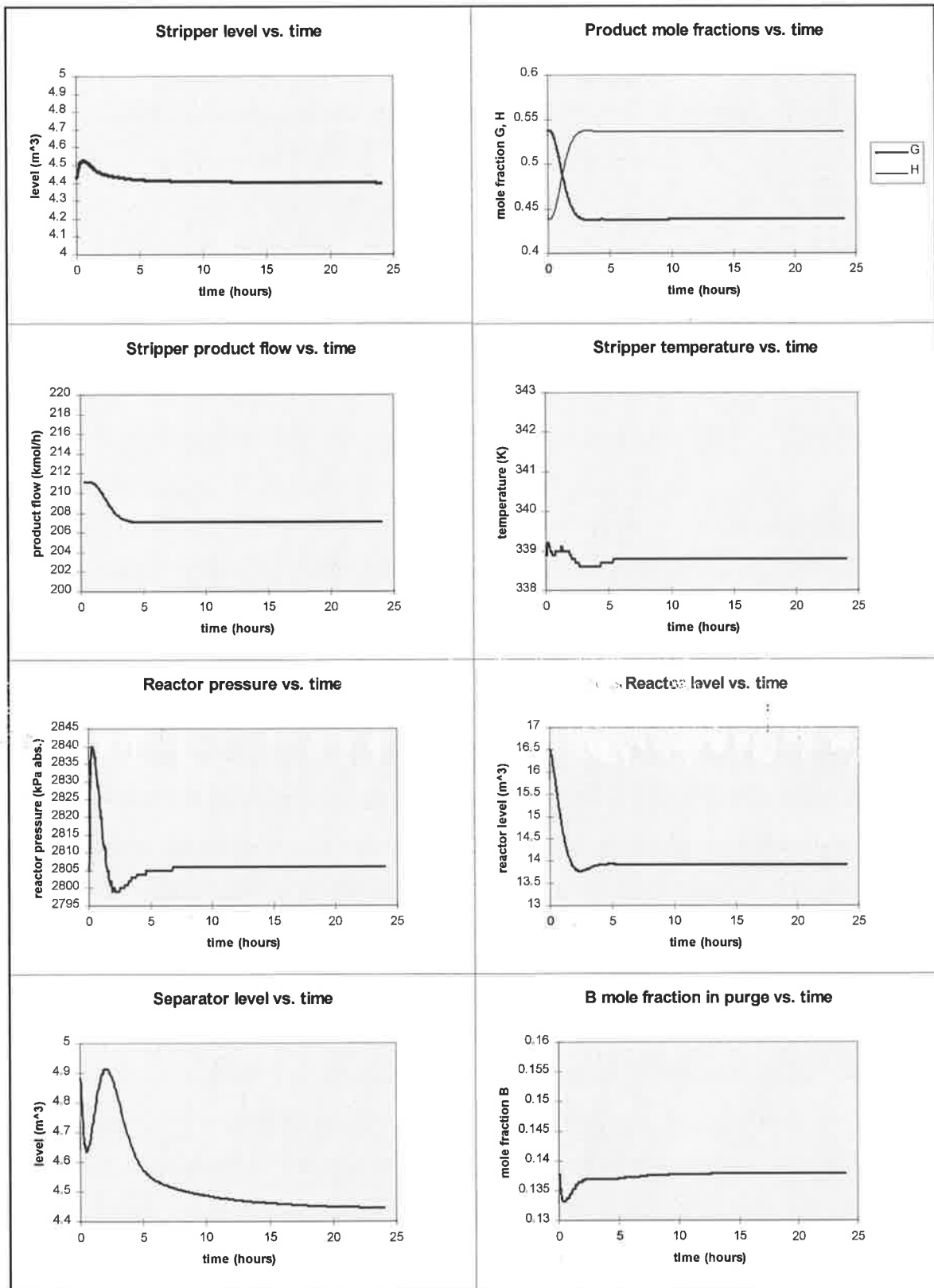
The purge setpoint responses are in Figure 5.9. The process has a large vapour volume and component B is a very minor part of the A and C feeds, so the setpoint change requires some time to propagate through the system. The new setpoint is reached after about 11 hours with little effect on the rest of the process.

The control system and plant model contained 203 equations. The four simulations were completed in about 2 hours each on a Pentium 100 MHz with 16 MB RAM. The BDF solver employed a non-sparse matrix technique and solved the dynamic and algebraic equations simultaneously.

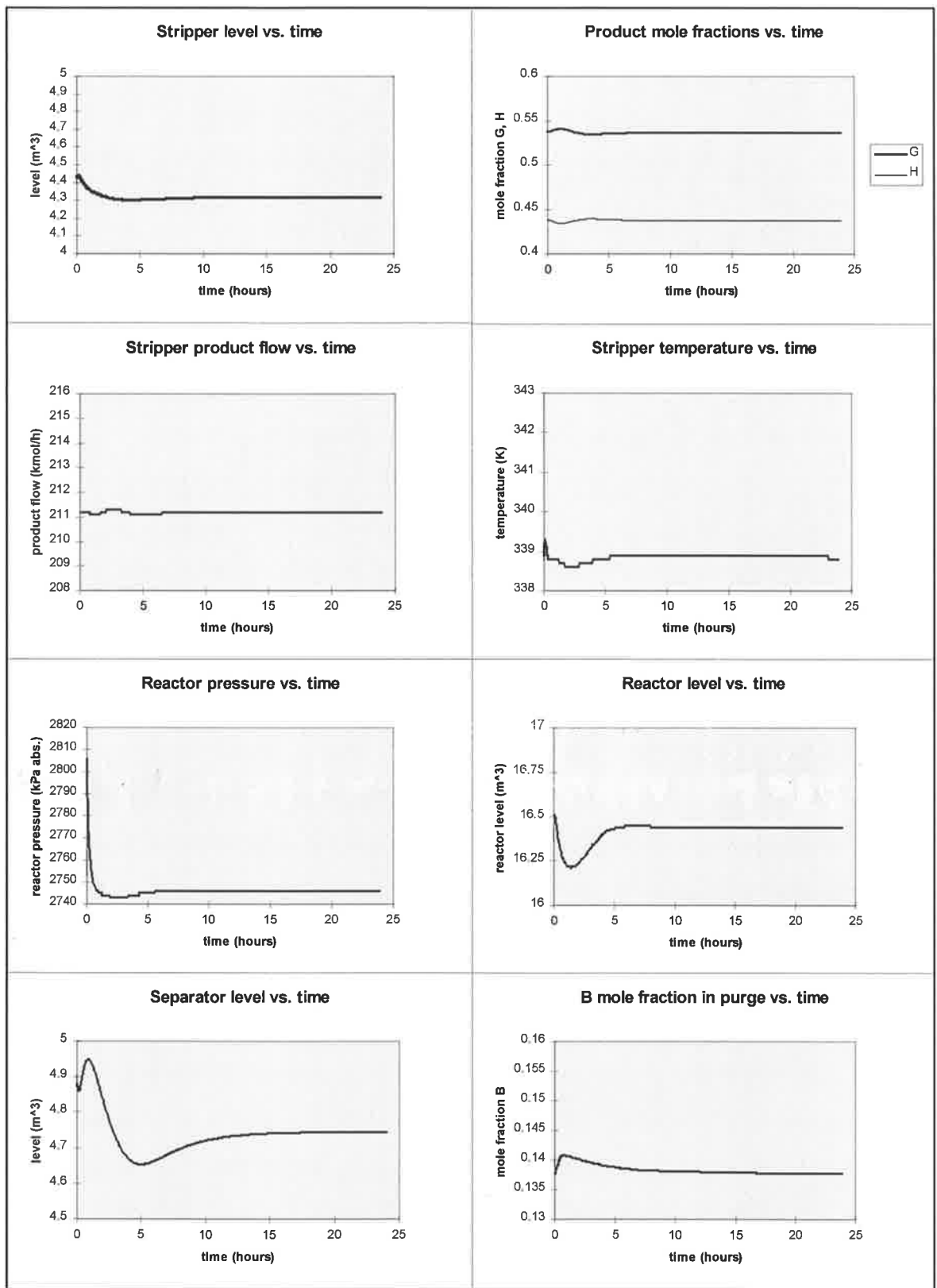
The complete Tennessee Eastman problem statement describes a number of plant upsets and disturbances. The paper by Luyben describes the (few) deficiencies of this control system and suitable overrides for coping with the upsets. The overrides were not incorporated into the control system described here and were not simulated. Overrides are generally event-driven, and the override controllers would need to be objects of a class that can take appropriate event-based action. The main difficulty would be providing sufficient functionality for the many different events a user will require. A similar approach to user-defined unit models with the **System** hierarchy may be taken. The existing **Port** hierarchy is sufficient for connecting to process variables. The `dynamic_model()`, `disc_check()` and `update()` functions could perform the event actions as required. The main difficulty would be the potential changes to the equation structure of the problem.



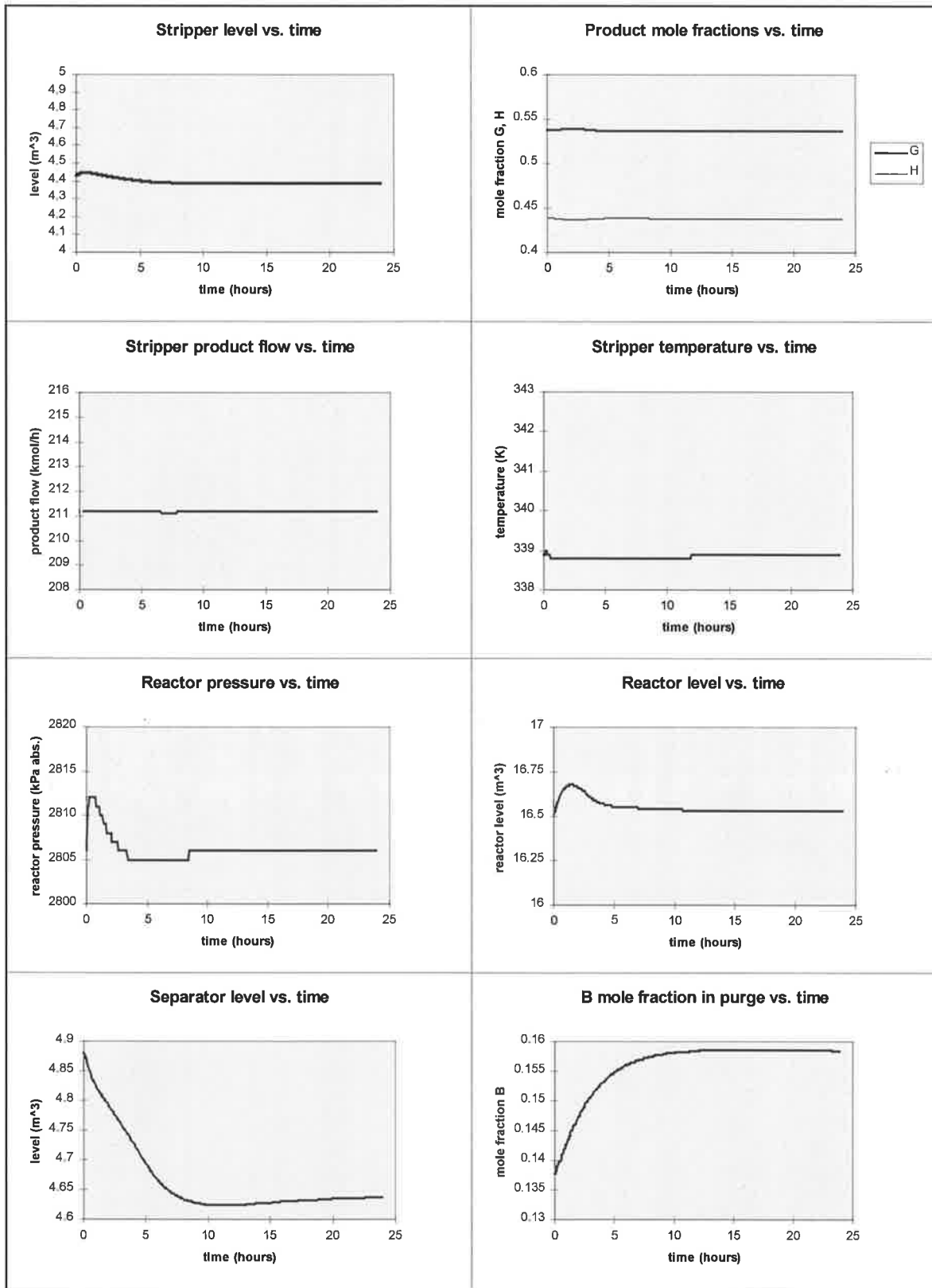
**Figure 5.6: Tennessee Eastman response to 15 % decrease in production rate.**



**Figure 5.7: Tennessee Eastman response to product G:H mass ratio setpoint change from 50:50 to 40:60.**



**Figure 5.8: Tennessee Eastman response to reactor pressure setpoint change from 2806 kPa abs. to 2746 kPa abs**



**Figure 5.9: Tennessee Eastman response to purge composition setpoint change from 13.8 mole % B to 15.8 mole % B.**

### 5.3 Recombinant Fermentation Model

For the duration of this project, the Chemical Engineering Department at Adelaide University has been a partner in a biochemical engineering research group. The Department's laboratory facilities include a plant for fermenting and purifying protein products from recombinant *Escherichia Coli* bacteria. The principal products from the plant are insulin-like growth factors (IGFs) for medical research. These growth factors are grown as inclusion bodies within cells that have been genetically modified by insertion of a plasmid into the cell's DNA. The plasmid is a piece of DNA that instructs the cell to produce the recombinant protein product. At the completion of the fermentation, cells are homogenised to release the inclusion bodies and are then centrifuged to recover the pure inclusion body protein.

As part of the project, it was decided to develop and simulate a complete recombinant fermentation model. The final model is based on first principles and is derived from various characteristics of recombinant *E. Coli* fermentation identified in the literature. No experimental work was undertaken. The model is not intended to be an exact representation of the departmental facilities, it was developed purely as a modelling and simulation exercise. The model is not specific to a particular recombinant protein product.

The general model form is based on Monod growth kinetics, substrate and product inhibition and assumption of an ideal stirred tank reactor as described in Nielsen and Villadsen (1991). An example of Monod kinetics is given below for growth of biomass  $X$  at a specific growth rate  $\mu$  on a substrate  $S$  with inhibition by a product  $P$ . The model describes growth, substrate consumption and product formation in a batch system of constant volume.

$$\frac{dX}{dt} = \mu X \quad (5.1)$$

$$\mu = \mu_{\max} \left( \frac{S}{S + K_S} \right) \left( 1 - \frac{P}{P_{\max}} \right) \quad (5.2)$$

$$\frac{dS}{dt} = -\frac{1}{Y_{XS}} \frac{dX}{dt} - \frac{1}{Y_{PS}} \frac{dP}{dt} - m_{XS} X \quad (5.3)$$

$$\frac{dP}{dt} = r_f(\mu) X \quad (5.4)$$

$X$ ,  $S$ ,  $K_S$ ,  $P$  and  $P_{\max}$  have units of g/L. Equation (5.1) is the dynamic biomass balance. Equation (5.2) describes the actual growth rate of the cells.  $K_S$  is a saturation constant for the dependence of growth rate on substrate concentration; the growth rate decreases rapidly when



$S$  is less than  $K_S$ .  $P_{\max}$  is the concentration of product at which growth ceases.  $\mu_{\max}$  is the maximum specific growth rate of the cells with units of  $h^{-1}$ . Equation (5.3) is the dynamic substrate balance.  $Y_{XS}$  and  $Y_{PS}$  are yields of biomass and product, respectively. Their units are (g biomass)/(g substrate) and (g product)/(g substrate).  $m_{XS}$  is the cells' maintenance requirement for metabolic activity, in (g substrate)/(g biomass)/h. Equation (5.4) is the dynamic product balance. The function  $r_{fp}(\mu)$  determines the rate of product formation. The units are (g product)/(g biomass)/h.

### 5.3.1 Model Description

The fermenters in the department normally operate in an aerobic, fed-batch mode. Therefore an overall volume (mass) balance must be incorporated into the basic model form above. Recombinant fermentations generally proceed without significant protein product formation until there is sufficient cell mass to produce a reasonable amount of product. Product formation is then initiated by an inducer, isopropylthiogalactoside (IPTG). The complete model is below:

$$\mu^+ = \mu_{\max}^+ \left(1 - \frac{P}{P_{\max}}\right) \left(1 - \frac{A}{A_{\max}}\right) \left(\frac{S}{S + K_S}\right) \left(\frac{O_2}{O_2 + K_{O_2}}\right) (k_s + k_r R_r) \quad /h \quad (5.6)$$

$$\mu^- = \mu_{\max}^- \left(1 - \frac{A}{A_{\max}}\right) \left(\frac{S}{S + K_S}\right) \left(\frac{O_2}{O_2 + K_{O_2}}\right) (k_s + k_r R_r) \quad /h \quad (5.7)$$

$$\frac{d(VX^+)}{dt} = (1 - p_r) \mu^+ (VX^+) \quad g \text{ bm} / h \quad (5.8)$$

$$\frac{d(VX^-)}{dt} = p_r \mu^+ (VX^+) + \mu^- (VX^-) \quad g \text{ bm} / h \quad (5.9)$$

$$\frac{d(VA)}{dt} = r_{f_0}(\mu^-) (VX^-) + r_{f_0}(\mu^+) (VX^+) \quad g \text{ acet} / h \quad (5.10)$$

$$\frac{d(VS)}{dt} = F_s - \frac{1}{Y_{XS}} \left(\frac{d(VX^+)}{dt} + \frac{d(VX^-)}{dt}\right) - \frac{1}{Y_{AS}} \frac{d(VA)}{dt} - \frac{1}{Y_{PS}} \frac{d(VP)}{dt} - m_{XS} (VX^+ + VX^-) \quad g \text{ gluc} / h \quad (5.11)$$

$$\frac{d(VP)}{dt} = r_{fp}(\mu^+) \left(\frac{f_I + I}{K_I + I}\right) (VX^+) \quad g \text{ prot} / h \quad (5.12)$$

$$\frac{d(VO_2)}{dt} = V k_1 a (O_2^* - O_2) - \frac{1}{Y_{XO_2}} \left(\frac{d(VX^+)}{dt} + \frac{d(VX^-)}{dt}\right) - m_{XO_2} (VX^+ + VX^-) \quad mmol O_2 / h \quad (5.13)$$

$$\frac{dV}{dt} = \frac{F_s}{\rho_s} \quad L / h \quad (5.14)$$

$$\frac{dk_s}{dt} = -k_1 k_s \quad (5.15)$$

$$\frac{dk_r}{dt} = k_1 (1 - k_r) \quad (5.16)$$

$$R_r = \frac{C_A}{C_A + I} \quad (5.17)$$

$$p_r = \lambda \left( \mu^+ - \frac{V^0 \left( \frac{\mu^+}{K_s} \right)^n}{1 + \left( \frac{\mu^+}{K_s} \right)^n} \right) \quad (5.18)$$

$$r_{fa}(\mu) = 0.7(\mu - 0.5) \quad \text{g acet / g bm / h} \quad (5.19)$$

$$r_{fp}(\mu^+) = K_0 \mathcal{E}(\mu^+ + 0.036) G_p(\mu^+) \quad \text{g prot / g bm / h} \quad (5.20)$$

$$G_p(\mu^+) = \frac{0.2}{\mu^+} + 0.1 \quad \text{mg plasmid / g bm} \quad (5.21)$$

$$k_{1a} = 3600 k_{u_s} \alpha \left( \frac{P_M}{V} \right)^\beta \quad /h \quad (5.22)$$

$$P_M = N_p \rho N^3 d_s^5 \quad W \quad (5.23)$$

$$O_2^* = \frac{P_{O_2}}{94.51} \quad \text{mmol / L} \quad (5.24)$$

$$P_{O_2} = 0.2095 (P_F - P_{H_2O}) \quad \text{kPa} \quad (5.25)$$

|                 |                                      |                              |
|-----------------|--------------------------------------|------------------------------|
| -               | plasmid-free biomass                 |                              |
| +               | plasmid-carrying biomass             |                              |
| $A$             | acetate concentration                | (g acetate) /L               |
| $F_S$           | glucose feed rate                    | (g glucose) / h              |
| $G_p(\mu^+)$    | plasmid concentration in biomass     | (mg plasmid) / (g biomass)   |
| $I$             | IPTG concentration                   | (g IPTG) /L                  |
| $k_{1a}$        | volumetric mass transfer coefficient | $h^{-1}$                     |
| $k_r$           | IPTG recovery rate variable          |                              |
| $K_S$           | saturation constant for glucose      | (g glucose) /L               |
| $k_s$           | IPTG shock rate variable             |                              |
| $\mu$           | specific growth rate                 | $h^{-1}$                     |
| $N$             | agitator rotational speed            | rev /s                       |
| $O_2$           | oxygen concentration                 | (mmol $O_2$ ) /L             |
| $O_2^*$         | saturated oxygen concentration       | (mmol $O_2$ ) /L             |
| $P$             | protein product concentration        | (g protein) /L               |
| $P_F$           | fermentor absolute pressure          | kPa                          |
| $P_M$           | mixing power                         | W                            |
| $P_{O_2}$       | oxygen partial pressure              | kPa                          |
| $p_r$           | probability of plasmid loss          |                              |
| $r_{fa}(\mu)$   | rate of formation of acetate         | (g acetate) / (g biomass) /h |
| $r_{fp}(\mu^+)$ | rate of formation of protein product | (g protein) / (g biomass) /h |
| $R_r$           | IPTG recovery ratio                  |                              |
| $S$             | glucose concentration                | (g glucose) /L               |
| $u_s$           | superficial gas velocity             | m /s                         |
| $V$             | broth volume                         | L                            |
| $X$             | biomass concentration                | (g biomass) /L               |

The model parameters are in Appendix E.

The effects of different factors are evident in several equations. Equations (5.6) and (5.7) describe the actual growth rates of the plasmid-harboring and plasmid-free cells. Both equations include terms for substrate and product inhibition. The values for glucose and oxygen saturation constants were calculated for *E. Coli* as recommended in Roels (1983). The inhibiting acetate concentration  $A_{\max}$  is extracted from Konstantinov *et al.* (1990). The term  $(k_s + k_p R_p)$  determines the effect of IPTG inducer on cell growth rate. *E. Coli* cell growth and product formation rate are both affected by the concentration of the inducing agent. The addition of IPTG “shocks” the bacteria and their growth rate slows. This effect has been modelled as a combination of time-differential shock and recovery terms by Lee and Ramirez (1992). It is assumed that the addition of IPTG affects cells with and without plasmids. These workers also modelled the IPTG concentration’s contribution to the protein production rate with a saturation expression similar to those for substrate saturation. To account for this the  $\left(\frac{f_i + I}{K_i + I}\right)$  expression is included in equation (5.12). The term  $f_i$  accounts for the fact that the recombinant cells produce small amounts of protein without any inducer present.

Equations (5.8) and (5.9) are the dynamic balances for plasmid-containing and plasmid-free cells, respectively. Plasmid-containing cells have a tendency to lose the plasmid as they replicate (Lee *et al.*, 1988), thereby reducing product yields. In very small-scale experiments, it is possible to incorporate a gene into the cells that renders the plasmid-containing cells immune to certain antibiotics, such as ampicillin or tetracycline. The antibiotic is then fed into the cell culture. Plasmid-free cells are not resistant to the antibiotic and die without replicating, thus ensuring a pure cell culture. This is known as selection pressure. On large-scale equipment it is prohibitively expensive to apply such selection pressure and hence the fermentation broth may contain a significant number of useless cells.

The probability of plasmid loss from *E. Coli* has been modelled by Mosrati *et al.* (1992). The probability term is  $p_r(\mu^+)$  in the equations and is calculated in equation (5.18). It is generally less than a few percent, however over the course of a fermentation the fraction of plasmid-free cells can increase significantly. This is demonstrated in the model simulations.

Equation (5.10) provides an acetate balance. Acetic acid production is also related to the cell growth rate. This has been modelled by Majewski and Domach (1990). The acetate

production rate  $r_{\mu}(\mu)$  is calculated in equation (5.19). Equation (5.11) is the glucose substrate balance, similar to the earlier equation (5.3). The yield and maintenance coefficients are calculated as per Roels (1983), except  $Y_{PS}$ .  $Y_{PS}$  is approximately calculated from data presented in Cockshott and Bogle (1992).

Equation (5.12) summarises the protein product balance. The dependency of protein product formation on the inducer concentration has been described above. Product formation is also dependent on the cell growth rate and the concentration of plasmid in the cell (Lee *et.al.* 1988). The rate of product formation  $r_{p}(\mu^{\dagger})$  is calculated in equations (5.20) and (5.21).

Equation (5.13) is the oxygen substrate balance. The oxygen yield and maintenance terms are calculated as per Roels (1983). It is assumed that protein product formation is part of the general cell oxygen yield and maintenance. The correlation for the volumetric mass transfer coefficient (equation (5.22) and impeller power requirement (equation (5.23)) is from Nielsen and Villadsen (1991)). Equations (5.24) and (5.25) calculate the saturated oxygen concentration from Henry's Law.

The volume balance for the fermenter is presented in equation (5.14).  $V$  is pure broth volume because the bubble phase is not included in the balance.

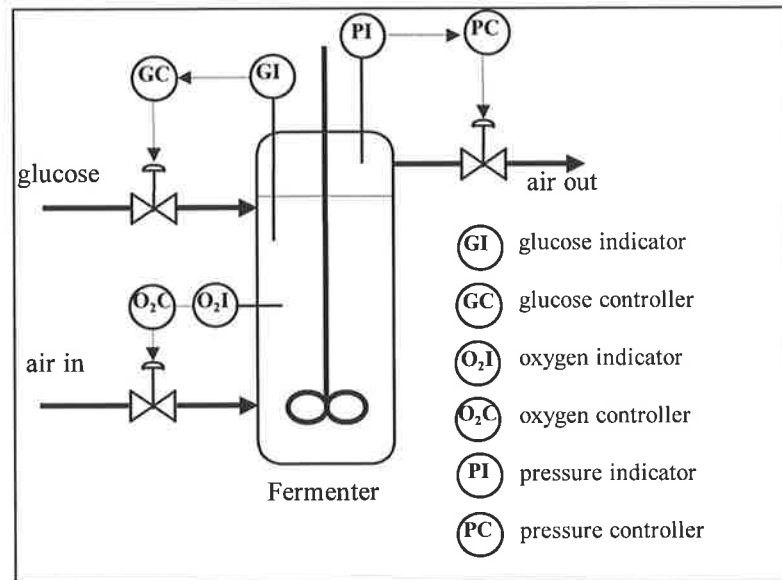
Finally, equations (5.15), (5.16) and (5.17) calculate the shock and recovery parameters for the IPTG growth effects.

Some further comments are in order. The literature cited above covers a variety of different *E. Coli* genetic strains, plasmid types and protein products. Therefore the model will not be an accurate description of any particular process. However, the model should reasonably simulate the general features of a recombinant fermentation.

### 5.3.2 Control System

A desirable goal of any fermentation of this type is to maximise the yield of recombinant protein product. An examination of the model indicates potential controlled variables in a process control scheme. The obvious controlled variables for this model are the substrate concentrations (i.e. glucose and oxygen). With one of these held in excess, control of the

other should regulate the cell growth rate, which in turn affects the rate of product and non-viable cell formation. Oxygen was selected to be the excess substrate, with glucose concentration as the major controlled variable. The final control scheme is illustrated below in Figure 5.10.



**Figure 5.10: Fermenter control system diagram**

The oxygen concentration is controlled by the regulation of air flow into the fermenter broth. The air flow determines the oxygen transfer rate  $k_L a$  in equation (5.13). The agitation speed is held constant. The pressure in the fermenter is controlled by regulating the outlet air flow. The glucose concentration is controlled by regulating the flow of a glucose solution to the fermenter.

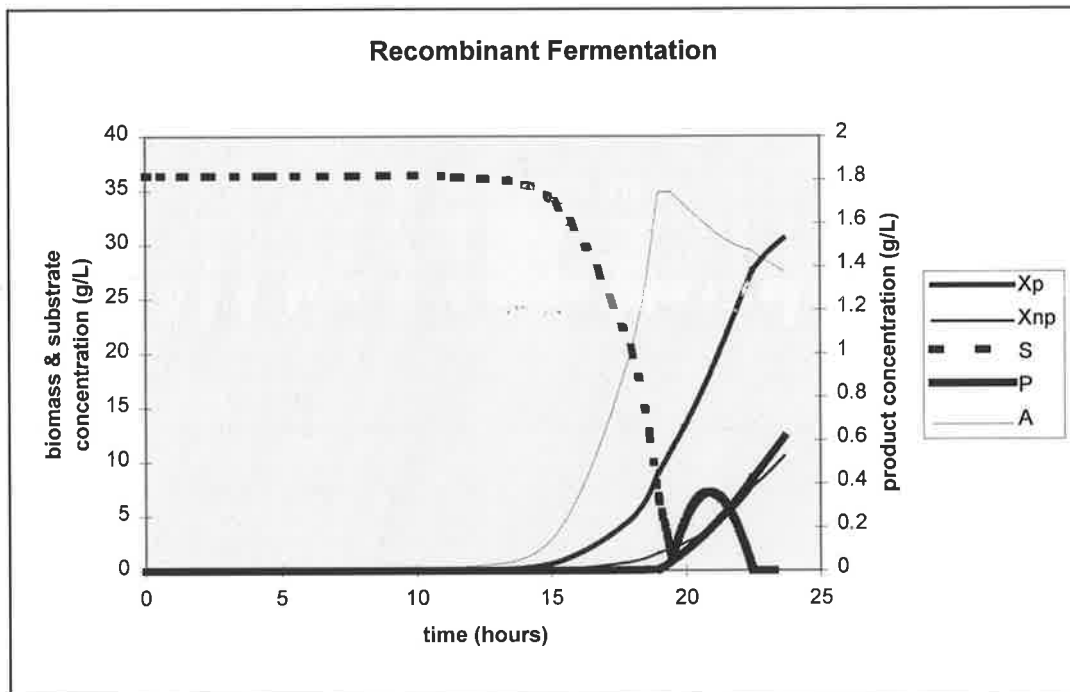
### 5.3.3 Simulation Results

The objective of the simulations was to investigate the differences between various glucose feed flows and controller setpoints. The simulations were run from a consistent basis:

- Initial inoculation of 250  $\mu\text{g}$  of plasmid-harboring cells into the fermenter.
- Dissolved oxygen concentration controlled to 40% of saturation, referenced to atmospheric pressure and 37°C.

- IPTG was added at a *total* biomass concentration of 10.0 g/L. At induction 1.1 g of IPTG was added in a 0.1 L solution to the broth.
- Fermenter pressure was controlled to 200 kPa absolute.
- Initial glucose concentration was 36.0 g/L in an initial broth volume of 15.0 L.
- A total of 4.5 L of glucose feed solution at a concentration of 300 g/L glucose was available for fed-batch operation.

As a comparison to a fully-controlled glucose feed, a simple on-off control system was also investigated. The on-off system was set up to add glucose to the broth at a constant 325 g/h (approximately 1.08 L/h feed solution) once the broth glucose concentration had decreased to 0.1 g/L. The results for the on-off control simulation are presented in Figure 5.11.

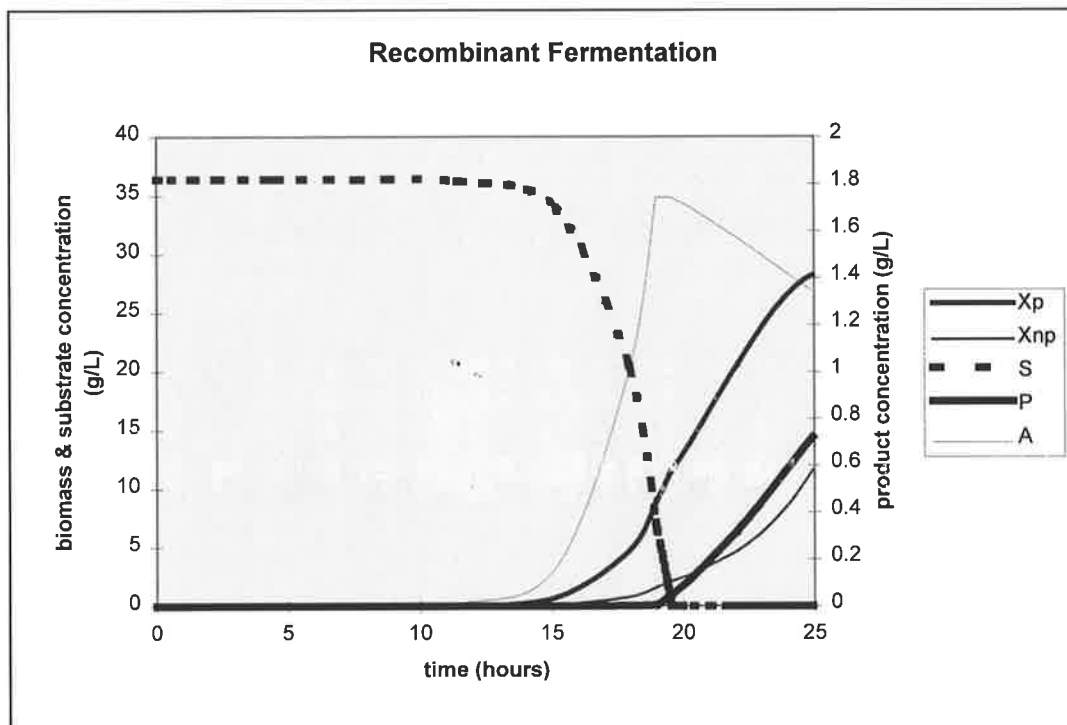


**Figure 5.11: On-off glucose control simulation.**

After about 15 hours the broth glucose concentration decreased rapidly as the cells entered an exponential growth phase. At approximately  $t = 18$  hours the broth was induced with IPTG. At  $t = 19.5$  hours the constant-rate glucose feed commenced and the glucose concentration briefly rose before the increasing biomass rapidly consumed the extra glucose. Just after 23 hours the glucose feed was exhausted and the simulation ceased. The final protein

concentration was 0.61 (g protein) /L. The specific protein yield was 0.0197 (g protein) /g plasmid-harboured cells. The final acetate concentration was 1.38 g /L, not enough to have significantly inhibited growth because the acetate inhibition constant is 15.0 g /L.

The response for a simulation with full glucose control, with the glucose controlled to a setpoint of 0.01 g /L is presented below in Figure 5.12. The glucose substrate saturation constant  $K_s$  is 0.004 g /L. 0.01 g /L glucose would therefore retard the growth rate by approximately 29% over an excess glucose system.

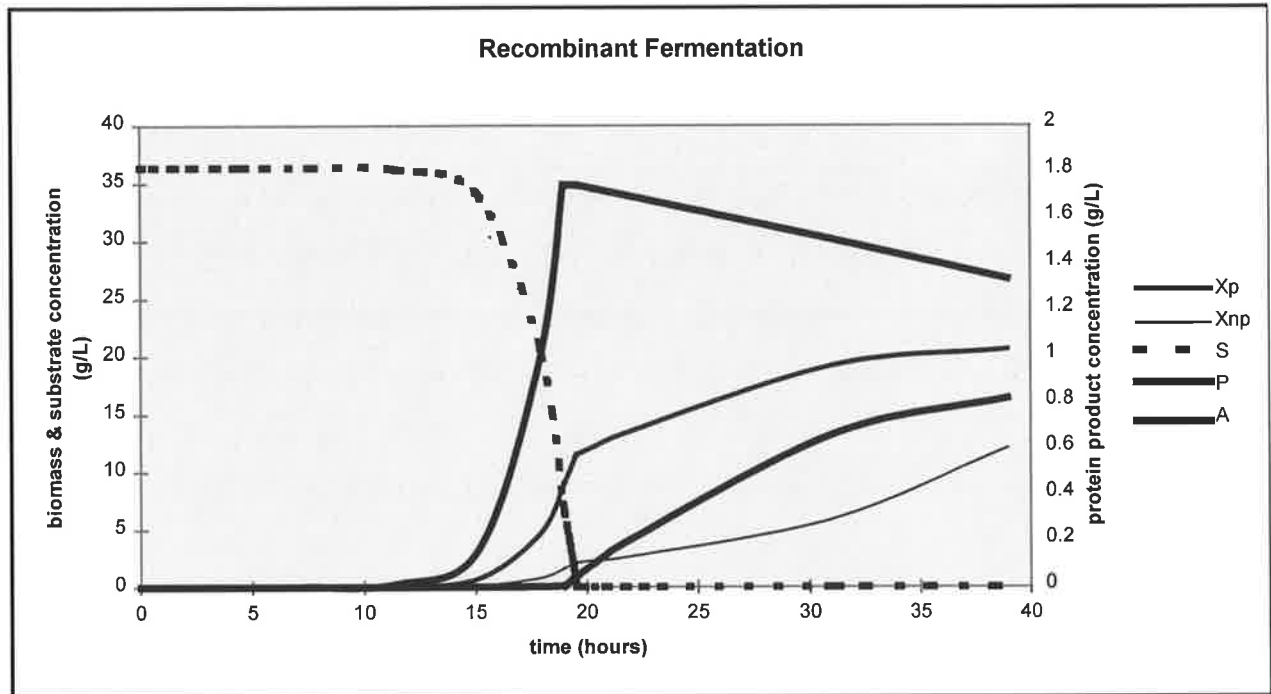


**Figure 5.12: Full glucose control simulation. Setpoint 0.01 g /L.**

No glucose hump is present in this run. The broth was induced at about 18 hours. The growth rate decreased very slightly at  $t = 19.5$  hours. The average glucose feed rate was slightly lower and so the run continued for 24 simulated hours before stopping. The extra production time resulted in a final protein concentration of 0.73 g /L. The specific protein yield was 0.0244 g / g. The final acetate concentration was 1.33 g /L.

A further run was undertaken to investigate the effects of a very low controlled glucose concentration of 0.001 g /L. A glucose concentration of 0.001 g/L would retard the growth

rate by about 80% over an excess glucose system. The simulation results are presented in Figure 5.13.



**Figure 5.13: Full glucose control simulation. Setpoint 0.001 g /L.**

At an equivalent 24 hours, the protein concentration is only around 0.4 g /L because of the restricted growth rate. However, the final protein yield was 0.81 g /L. Equation (5.21) indicates that at lower growth rates, the plasmid concentration in the biomass is greatly increased. The specific protein yield was 0.0403 g /g, considerably higher than the previous runs. The comparatively low glucose feed rate meant that protein production could continue for 12 hours longer than the other runs, providing a higher production efficiency in spite of the retarded growth rate.

#### **5.4 Discussion**

The advantages of integrated steady-state and dynamic process simulation become apparent when applied to complex test problems. With a sufficiently accurate model and a suitable simulator, the performance of a plant may be determined without disturbing the physical process. This is particularly well demonstrated by the Tennessee Eastman model. Well over 100 simulation runs were undertaken while evaluating and tuning the various control schemes



discussed. This would be prohibitively expensive in production costs, safety risk and time on the physical plant.

A process model can provide information about process variables that are not physically measured, simply by interrogating the appropriate unit operation model. The degree of confidence in the simulated information is of course dependent on the accuracy of the model.

Dynamic simulation provides a great deal more information about plant operation than steady-state simulation alone. The Cavett problem was originally designed as a sequential-modular convergence test problem but a dynamic analysis is still applicable. The simulated steady-state design specification on *i-butane* recovery provides an accurate operating point for the temperature of the second flash unit only when the compositions of the feed streams and the other unit conditions are exactly as simulated. Changes in process conditions must be re-simulated to determine a valid operating range. Alternatively, dynamic simulation offers the ability to implement and test control system designed to maintain the *i-butane* recovery over a wide range of process conditions. Hydrocarbon systems are well-understood and the results may be expected to be quite accurate. Steady-state design tools for control systems cannot provide the same information. One major benefit of steady-state simulation is its effectiveness for initialising a dynamic simulation. Even the fermentation model was initialised with steady-state simulation, to obtain controller outputs and setpoints.

The Tennessee Eastman problem statement specifies a wider range of process operating conditions than those explored here. The steady-state plant model may be simulated to provide any of the desired plant conditions at a wide variety of individual unit conditions. For example, the unsuccessful pressure-temperature control system may be solved for all of the operating conditions in the original paper. This does not provide any information about the dynamic performance however.

Finally, integrated steady-state and dynamic simulation aids process understanding. The descriptions of the behaviour of the unsuccessful control schemes were determined from sets of simulation results. The Tennessee Eastman plant model is quite complex and as such is effectively useless without some form of simulation to describe the model's behaviour.

## 5.5 Summary

Three processes were employed to evaluate the simulator performance and applicability. The different steady-state capabilities of the simulator were examined with the four-flash Cavett problem. The unit models were the same in all simulations. No simulation technique was found to be completely superior. Sequential-modular process simulation is a useful technique for initialising equation-oriented simulations and can be readily applied to design problems if the unit models are solved in an equation-oriented form.

The dynamic capabilities of the simulator were investigated with the Tennessee Eastman Challenge Problem. The dynamic system was initialised with equation-oriented steady-state solution of the dynamic unit models. Several control systems were evaluated with varying degrees of success. The capability to break connections between **Variable** objects was demonstrated to be useful for simulating saturated controllers. Complete results for the very simple and effective control scheme proposed by Luyben (1996) were presented. The process remained under stable control for the process changes initiated.

Finally a recombinant fermentation model was developed and simulated. The model does not represent a specific process and may not be generally applicable. Different controlled glucose feeding strategies were demonstrated to increase product yield by approximately 50 % over simple uncontrolled systems.

The next chapter is the final chapter of the thesis and summarises the project's conclusions and suggests directions for future work.

# CHAPTER 6

## Summary, Conclusions and Recommendations

### 6.1 Summary

The preceding chapters have discussed the requirements, design, implementation and testing of a C++ class structure for simulation of biochemical and chemical processes. The simulator is capable of steady-state and dynamic simulation, employing the same unit models. Multiple, interchangeable steady-state simulation techniques are supported. The data structure is designed to promote user-defined process models. The project has met the objectives outlined in Chapter 1.

### 6.2 Class Description

An examination of the physical attributes of a process flowsheet provided the basis for the design of a class structure to represent general flowsheet objects and their connections. The design of the mathematical structure followed from the requirement of multiple solution methods, both steady-state and dynamic. A versatile class structure has been developed, summarised below into four main areas:

1. Process representation classes for modelling the physical attributes of a chemical process. There are three main parent classes in this group: **System**, for modelling entities that transform information or material, **Port**, for defining connection interfaces to **Systems**, and **Streams**, for connecting **Systems** through their **Ports**.
2. Mathematical representation classes for modelling the mathematical structures contained within **System**-types. There are two main parent classes in this group: **Equation\_Set** for collecting sets of model equations and **Variable** for modelling the individual components of an equation.
3. Physical property classes for modelling the behaviour of chemical components and mixtures manipulated by **System**-types. There are three main parent classes in this group: **Component**, for modelling individual chemical components,

**General\_Component\_Mixture**, for modelling mixture phases and **Properties**, for calculating the properties associated with a mixture or phase.

4. Numerical method classes to provide solution methods for sets of equations. This group of classes has comparatively little structure and is designed to provide solution functionality. The class structure is based on single inheritance, commencing with a **Math\_Top** class, through **Math\_Util**, **Linear\_Solver**, **Nonlinear\_Solver**, **DAE\_Solver** and **Mathtool** classes.

The capability for **Systems** to contain other **Systems** and **Equation\_Sets** to contain other **Equation\_Sets** facilitates model description because the executive-level structure of a complex flowsheet has the same basis as the executive-level structure of a simple valve. The containment principle creates a readily-analysed, connected tree of System objects and **Equation\_Set** objects which in turn permits a variety of solution techniques to be applied. The **Sys\_Man\_Block** class exploits this capability to cater for connected multi-**System** models as complex unit operations or as flowsheets. Different sections of a **Flowsheet** may be examined and solved independently of the rest by allocating the sections to a **Sys\_Man\_Block** object. The sections may overlap. The management of **System**-types is demonstrated by the solution method controls built in to the **Convergence\_Block** class.

### 6.3 Modelling

The classes provide a consistent framework for model definition and solution. Various model decomposition techniques have been described: Medium and Machine Decomposition, Primitive Behaviour Decomposition and Mathematical Decomposition. The application of the techniques with the class structure was discussed.

The class structure supports bi-directional information flow. This was demonstrated with a **Control\_Valve** class. Models may acquire attributes through aggregation and connection of sub-objects or through multiple inheritance of characteristics. These were demonstrated with two separate definitions of a **Ratio\_Controller** class. Aggregation provided a stricter modelling methodology that did not break the encapsulation of the sub-objects and in particular left the final class with a single set of accessible interface functions. Multiple inheritance of characteristics left some ancestor interfaces still functional. This can corrupt the **System**-level structure that the rest of the simulator interacts with unless suitable

precautions are taken. With this class structure, multiple inheritance modelling requires more knowledge about the parent classes and more effort to ensure consistent object construction. The physical property classes were demonstrated with a multicomponent **Flash** model class. A three-component flash was simulated in steady-state to introduce **Flowsheet**-level simulation.

The permitted level of access to internal attributes of an object is an interesting issue in software engineering. As stated in previous chapters, the majority of the **System**-level structure must be inaccessible from specific unit-operation objects. This preserves the consistency of the structure at an operational level. However, a system in which unit model definition is in the same language as the important high-level data structure and functionality is potentially easier to corrupt, especially in a system operated through a compiler. The low-level model attributes are nearly always accessible; this could be considered a necessity in an environment that encourages user-defined models and versatility. A system for widespread use could perhaps supply the high-level code as a precompiled set of libraries that are automatically attached to the user's low-level code. This would seem to be a simple option, the concept of Dynamic-Linked-Libraries (DLLs) is popular in C++. Complete resolution of the issue is not required at this stage of the project's development.

#### **6.4 Simulation**

Three test processes were simulated; the four-flash Cavett problem, a recombinant fermentation model and the Tennessee Eastman control challenge problem. The simulator provides interchangeable sequential-modular, parallel-modular and equation-oriented simulation techniques with the same unit models. This was demonstrated with the Cavett problem. Sequential-modular simulation was especially effective for initialising equation-oriented simulations, although no single flowsheeting or numerical method proved to be completely superior. Sequential- and parallel-modular design problems were easily specified and solved with equation-oriented unit model solution.

A biochemical application was demonstrated with the development of a moderately complex recombinant fermentation model for protein production in *E. Coli* bacteria. The model was used to investigate potential advantages of process control applied to glucose feed flow.

The major test process for the dynamic capabilities of the simulator was the Tennessee Eastman problem. The original Fortran code was reverse-engineered into a set of unit operation classes. The physical properties were calculated with the ideal property and **User\_Component** classes in the simulator. Equation-oriented steady-state simulation initialised the dynamic system. The same unit models were employed for both steady-state and dynamic simulation. The control system was capable of adjusting the process operation rapidly to suit the four standard setpoint changes for the plant. The real-time-to-simulated-time ratio of the control system tests was about 1:10. There were 203 equations and variables.

### 6.5 Recommendations

Further development of the simulator is justified if emphasis is placed on modelling and simulation capabilities for small, unconventional processing systems. There are many packages available for large-scale simulation of conventional systems. The recommendations below reflect this special-purpose focus.

A better interface is required to facilitate the modelling process, especially if the emphasis is placed on special-purpose modelling. At this stage, model development is restricted to the text-based compiler environment with C++ code. The simulator output is text-based, either to the screen or to a file. Dynamic simulation results must be read into a spreadsheet program for plotting. The addition of real-time graphing facilities would greatly enhance the simulator.

The physical property package is quite basic. Only ideal and Peng-Robinson methods are available. An object-oriented physical property package is probably a thesis in itself. Physical property calculation is generally a service function to a simulation package. A comprehensive interface to an existing property package would be a more sensible and much simpler extension project than a complete object-oriented property database.

The data structure should be extended to handle other material types, for example solids or slurry processing. The basic **Process\_Port** class could be a starting point, because the contained **Variables** are not of a specific type. The composition **Vector** in a **Process\_Port** could be employed as fractions within a size range. A slurry characterisation would greatly

expand the capabilities for bioprocess modelling simulation, such as centrifugation or homogenisation operations.

The multiple steady-state solution techniques add complexity to the simulator. While the mathematical superiority of the equation-oriented approach is recognised, the difficulty associated with commencing the solution of larger problems (specifically the provision of reasonable initial estimates) requires a simple, robust initialisation technique. The basic sequential-modular capability should be retained for the initialisation.

Finally, a robust, sparse-matrix solver is recommended for equation-oriented simulation. The Tennessee Eastman Jacobian matrix is almost 98% sparse, for example. The sparse solver is probably better provided by an interface to existing third-party numerical software, because the interface coding is likely to be simpler than the coding of another numerical method.

# BIBLIOGRAPHY

- Acton, F.S. 1990. *Numerical Methods That Work*. 454-458. Mathematical Association of America, Washington.
- Ballinger, G.H., Bãnares-Alcántara, R., Costello, D., Fraga, E.S., Krabbe, J., Lababidi, H., Laing, D.M., McKinnel, R.C., Ponton, J.W., Skilling, N., and Spenceley, M.W. 1994. *épée: a Process Engineering Software Environment*. In *Proceedings of European Symposium on Computer-Aided Process Engineering- 3 (ESCAPE-3)*. Suppl. *Computers and Chemical Engineering*, **18**, S283-S287.
- Banerjee, A. and Arkun, Y. 1995. Control Configuration Design Applied to the Tennessee Eastman Plant-Wide Control Problem. *Computers and Chemical Engineering*, **19** (4), 453-480.
- Biegler, L.T. 1983. Simultaneous Modular Simulation and Optimisation. In *Proceedings of the 2<sup>nd</sup> International Conference on Foundations of Computer-Aided Process Design*, Westerberg, A.W. and Chien, H.H., (eds.), 369-408. Cache, Ann Arbor, Michigan.
- Bischak, D.P., and Roberts, S.D. 1991. Object-oriented Simulation. In *Proceedings of the 1991 Winter Simulation Conference*, B.L. Nelson, W.D. Kelton and G.M. Clark (eds.), 194-203. Institute of Electrical and Electronic Engineers, San Francisco, California.
- Bogle, I.D.L., and Perkins, J.D. 1988. Sparse Newton-Like Methods in Equation Oriented Flowsheeting. *Computers and Chemical Engineering*, **12** (8), 791-805.
- Britt, H.I. 1980. Multiphase Stream Structures in the ASPEN Process Simulator. In *Proceedings of the 1<sup>st</sup> International Conference on Foundations of Computer-Aided Process Design*, Mah, R.H., (ed.), 471-510. Engineering Foundation, New York.
- Broyden, C.G. 1965. *Mathematics of Computation*, **19**, 577-593.
- Chung, Y., and Westerberg, A.W. 1990. A Proposed Numerical Algorithm for Solving Nonlinear Index Problems. *Industrial Engineering Chemistry Research*, **29** (7), 1234-1239.
- Cockshott, A.R. and Bogle, I.D.L. 1992. Modelling a Recombinant *E.Coli* Fermentation Producing Bovine Somatotropin. In *Modeling and Control of Biotechnical Processes 1992 and Computer applications in fermentation technology (5th International Conference) : selected papers from the IFAC and ICCAFT 5 meetings*, Keystone, Colorado, USA, 29 March - 2 April 1992. Kamin, M.Z. and Stephanopoulos, G., (eds.), Pergamon Press, New York, 219 - 222.
- Dahl, O.J., Myhrbaug, B., and Nygaard, K. 1968. *SIMULA-67 Common Base Language*. Norwegian Computing Centre, report no. S-2.



- Downs, J.J., and Vogel, E.F. 1993. A Plant-Wide Industrial Control Problem. *Computers and Chemical Engineering*, **17** (3), 245-255.
- Ellis, M.A., and Stroustrup, B. 1994. *The Annotated C++ Reference Manual*. Addison-Wesley, Massachusetts.
- Evans, L.B. 1988. Bioprocess Simulation: A New Tool for Process Development. *Bio/Technology*, **6** (2), 200-203.
- Evans, L.B., Boston, J.F., Britt, H.I., Gallier, P.W., Gupta, P.K., Joseph, B., Mahalec, V., Ng, E., Seider, W.D., and Yagi, H. 1979. ASPEN: An Advanced System for Process Engineering. *Computers and Chemical Engineering*, **3**, 319-327.
- Farza, M., and Chérury, A. 1991. CAMBIO: software for modelling and simulation of bioprocesses. *CABIOS*, **7** (3), 327-336.
- Fletcher, J.P., and Ogbonda, J.E. 1988. A Modular Equation-Oriented Approach to Dynamic Simulation of Chemical Processes. *Computers and Chemical Engineering*, **12** (5), 401-405.
- Fraga, E.S., McKinnon, K.I.M., and Johns, W.R. 1991. Process Synthesis using a Parallel Computer. In *Computer-Oriented Process Engineering*, L. Puigjaner and A. Espuña (eds.), 235-240. Elsevier, Amsterdam.
- Franks, R.G.E. 1972. *Modelling and Simulation in Chemical Engineering*. Wiley Interscience, New York.
- Gadjaru, V.V. 1992. *Development of a Process Simulator using Object-Oriented Programming: Information Modeling and Program Structure*. PhD Thesis, Chemical Engineering Department, Iowa State University.
- Gear, C.W. 1971. Simultaneous Numerical Solution of Differential-Algebraic Equations. *IEEE Transactions on Circuit Theory*, **CT-18** (1), 89-95.
- Gear, C.W. 1988. Differential-Algebraic Equation Index Transformations. *SIAM Journal of Scientific and Statistical Computing*, **9** (1), 39-47.
- Habchi, G., and Deloule, F. 1992. Study of modelling and simulation for a chemical production system. *Simulation*, **58** (6), 366-374.
- Hall, G. and Watt, J.M. 1976. *Modern Numerical Methods for Ordinary Differential Equations*. Clarendon Press, Oxford, London.
- Henley, E.J and Rosen. 1969. *Material and Energy Balance Computations*. Wiley and Sons.
- Hillestad, M., and Hertzberg, T. 1986. Dynamic Simulation of Chemical Engineering Systems by the Sequential Modular Approach. *Computers and Chemical Engineering*, **10** (4), 377-388.

- Hillestad, M., and Hertzberg, T. 1988. Convergence and Stability of the Sequential Modular Approach to Dynamic Process Simulation. *Computers and Chemical Engineering*, **12** (5), 407-414.
- Holl, P., Marquardt, W., and Gilles, E.D. 1988. DIVA - A Powerful Tool for Dynamic Process Simulation. *Computers and Chemical Engineering*, **12** (5), 421-426.
- Hutchison, H.P., Jackson, D.J., and Morton, W. 1986a. The Development of an Equation-Oriented Flowsheet Simulation Package - I. The Quasilin Program. *Computers and Chemical Engineering*, **10** (1), 19-29.
- Hutchison, H.P., Jackson, D.J., and Morton, W. 1986a. The Development of an Equation-Oriented Flowsheet Simulation Package - II. Examples and Results. *Computers and Chemical Engineering*, **10** (1), 31-47.
- Konstantinov, K., Kishimoto, M., Seki, T. and Yoshida, T. 1990. A Balanced DO-Stat and Its Application to the Control of Acetic Acid Excretion by Recombinant *Escherichia Coli*. *Biotechnology and Bioengineering*, **36**, 750-758.
- Kröner, A., Holl, P., Marquardt, W., and Gilles, E.D. 1990. DIVA - An Open Architecture for Process Simulation. *Computers and Chemical Engineering*, **14** (11), 1289-1295.
- Lambert, J.D. 1991. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*. Ch. 6, pp. 213-260, Ch. 7, pp. 261-284. John Wiley and Sons, Chichester.
- Lau, K.H. 1992. *Development of a Process Simulator using Object-Oriented Programming: Numerical Procedures and Convergence Studies*. PhD Thesis, Chemical Engineering Department, Iowa State University.
- Lee, H.H., and Arora, J.S. 1991a. Object-oriented Programming for Engineering Applications. *Engineering with Computers*, (7), 225-235.
- Lee, J. and Ramirez, W. 1992. Mathematical Modeling of Induced Foreign Protein Production by Recombinant Bacteria. *Biotechnology and Bioengineering*, **39**, 635-646.
- Lee, S.B., Ryu, D.D.Y., Seigel, R. and Park, S.H. 1988. Performance of Recombinant Fermentation and Evaluation of Gene Expression Efficiency for Gene Product in Two-Stage Continuous Culture System. *Biotechnology and Bioengineering*, **31**, 805-820.
- Lee, T.Y. 1991b. *The Development of an Object-Oriented Environment for the Modeling of Physical, Chemical and Biological Systems*. PhD Thesis, Chemical Engineering Department, Texas A&M University.
- Lefkopoulos, A., and Stadtherr, M.A. 1993. Index Analysis of Unsteady-State Chemical Process Systems - I. An Algorithm for Problem Formulation. *Computers and Chemical Engineering*, **17** (4), 399-413.

- Lu, Y., Clarkson, A., Titchener-Hookner, N., Pantiledes, C., and Bogle, D. 1994. Simulation as a Tool in Process Design and Management for Production of Intracellular Enzymes. *Transactions of the IChemE*, **72**, Part A, May, 371-375.
- Luyben, W.L. 1996. Simple Regulatory Control of the Eastman Process. *Industrial Engineering Chemistry Research*, **35**, 3280-3289.
- Lyman, P.R. and Georgakis, C. 1995. Plant-Wide Control of the Tennessee Eastman Problem. *Computers and Chemical Engineering*, **19** (3), 321-331.
- Majewski, R.A. and Domach, M.M. 1990. Simple Constrained-Optimisation View of Acetate Overflow in *E. Coli*. *Biotechnology and Bioengineering*, **35**, 732-738.
- Marquardt, D.W. 1963. *Journal of the Society for Industrial and Applied Mathematics*, **11**, 431-441.
- Marquardt, W. 1991. Dynamic Process Simulation - Recent Progress and Future Challenges. In *Proceedings of the 4<sup>th</sup> International Conference on Chemical Process Control*, Y. Arkun and W.H. Ray (eds.), 131-180. AIChE publication 67.
- Marquardt, W. 1993. An Object-oriented Representation of Structured Process Models. In *Proceedings of the European Symposium on Computer-Aided Process Engineering - I (ESCAPE-1)*. Suppl. *Computers and Chemical Engineering*, **16**, S329-S336.
- Mattsson, S.E., Andersson, M., and Åström, K.J. 1993. Object-Oriented Modelling and Simulation. In D.A. Linkens (ed.), *CAD for Control Systems*, pp. 31-69, Marcel Dekker, Inc, New York.
- McAvoy, T.J. and Ye, N. 1994. Base Control for the Tennessee Eastman Problem. *Computers and Chemical Engineering*, **18** (5), 383-413.
- Meyer, B. 1992. *Eiffel: The Language*. Prentice-Hall.
- Morris, R.C. 1992. Process Simulation: Successes and Failures. In *Proceedings of the 1992 Winter Simulation Conference*, J.J. Swain, D. Goldsman, R.C. Crain and J.R. Wilson (eds.), 1249-1255. Institute of Electrical and Electronic Engineers, San Francisco, California.
- Motard, R.L. 1989. Integrated Computer-Aided Process Engineering. *Computers and Chemical Engineering*, **13** (11/12), 1199-1206.
- N.E.L. 1982. *PPDS (Physical Property Data System) User Manual*. National Engineering Laboratory, United Kingdom.
- Nielsen, J. and Villadsen, J. 1991. Bioreactors: Description and Modelling. In *Biotechnology*, Vol. 3, Ch 2. Rehm, H.J., Reed, G., Puhler, A. and Stadler, P., (eds.). VCH, New York, 79 - 102.
- Nilsson, B. 1993. *Object-Oriented Modeling of Chemical Processes*. PhD Thesis, Department of Automatic Control, Lund Institute of Technology.

- Oh, M., and Pantiledes, C.C. 1994. A Modeling System for Lumped and Distributed Parameter Processes. In *Proceedings of the 1994 IChemE Research Event*, P.A. Shamfou, A.R.H. Cornish, L.S. Hershenbaum, S. Moore, N. Titchener-Hooker (eds.), v2, 791-793. IChemE, United Kingdom.
- Paloschi, J.R. 1996. Using Sparse Bounded Homotopies in Steady-State Simulation Packages. In *Proceedings of the European Symposium on Computer-Aided Process Engineering - 6 (ESCAPE-6)*. Suppl. *Computers and Chemical Engineering*, **20**, S285-S290.
- Pantiledes, C.C. 1988. SPEEDUP: Recent Advances in Process Simulation. *Computers and Chemical Engineering*, **12** (7), 745-755.
- Pantiledes, C.C. 1988. The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal of Scientific and Statistical Computing*, **9** (2), 213-231.
- Pantiledes, C.C. and Barton, P.I. Equation-Oriented Dynamic Simulation: Current Status and Future Perspectives. In *Proceedings of the European Symposium on Computer-Aided Process Engineering- 2 (ESCAPE-2)*. Suppl. *Computers and Chemical Engineering*, **17**, S263-S285.
- Patterson, G.K., and Rozsa, R.B. 1980. DYNOSYL: A General-Purpose Dynamic Simulator for Chemical Processes. *Computers and Chemical Engineering*, **4**, 1-20.
- Pegden, C.D., Shannon, R.E., and Sadowski, R.P. 1990. *Introduction to Simulation using SIMAN*. McGraw-Hill, New York.
- Perkins, J.D., and Sargent, R.W.H. 1982. SPEEDUP: A Computer Program for Steady-State and Dynamic Simulation and Design of Chemical Processes. AIChE Symposium Series No. 214, **78**, 1-11.
- Petrides, D. 1994. BioPro Designer: An Advanced Computing Environment for Modeling and Design of Integrated Biochemical Processes. In *Proceedings of the European Symposium on Computer-Aided Process Engineering - 3 (ESCAPE-3)*. Suppl. *Computers and Chemical Engineering*, **18**, S621-S625.
- Petrides, D., Cooney, C.L., Evans, L.B., Field, R.P., and Snoswell, M. Bioprocess Simulation: An Integrated Approach to Process Development. *Computers and Chemical Engineering*, **13** (4/5), 553-561.
- Petrides, P.P., and Cooney, C.L. 1993. *Trends in Biotechnology*, **11**.
- Piela, P.C., Epperly, T.G., Westerberg, K.M., and Westerberg, A.W. 1991. ASCEND: An Object-oriented Computer Environment for Modeling and Analysis: The Modeling Language. *Computers and Chemical Engineering*, **15** (1), 53-72.
- Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. *Numerical Recipes in C*. 2<sup>nd</sup> edition. Cambridge University Press, Aust.

- Pritsker, A.A.B. 1986. *Introduction to Simulation and SLAM II*, 3<sup>rd</sup> ed. Halsted Press, New York.
- Reid, R.C., Prausnitz, J.M. and Poling, B.E. 1987. *The Properties of Gases and Liquids*. McGraw-Hill, New York.
- Ricker, N.L. and Lee, J.H. 1995a. Nonlinear Model Predictive Control of the Tennessee Eastman Challenge Process. *Computers and Chemical Engineering*, **19** (9), 961-981.
- Ricker, N.L. and Lee, J.H. 1995b. Nonlinear Modeling and State Estimation for the Tennessee Eastman Challenge Process. *Computers and Chemical Engineering*, **19** (9), 983-1005.
- Roels, J.A. 1983. *Energetics and Kinetics in Biotechnology*. Elsevier Biomedical Press, Amsterdam.
- Sargent, R.W.H. 1981. A Review of Methods for Solving Nonlinear Algebraic Equations. *Foundations of Computer-Aided Chemical Process Design*, R.H. Mah, W.D. Seider (eds.), v1, 27 - 76. Engineering Foundation, New York.
- Schriber, T.J. 1991. *An Introduction to Simulation using GPSS/H*. John Wiley and Sons, New York.
- Seider, W.D., and Brengel, D.D. 1991. Nonlinear Analysis in Process Design. *AIChE Journal*, **37** (1), 1-38.
- Shacham, M. 1985. Comparing Software for the Solution of Systems of Nonlinear Equations Arising in Chemical Engineering. *Computers and Chemical Engineering*, **9** (2), 103-112.
- Siletti, C.A. 1990. Design of Protein Purification Recovery Processes. In *Artificial Intelligence in Process Engineering*, M.L. Mavrouniotis (ed.), 295-310, Academic Press.
- Simon, F., Narosdoslawsky, M., Csermely, Z., and Altenburger, J. 1994. Physical Property Data Management in a Bioprocess Simulation System. In *Proceedings of the European Symposium on Computer-Aided Process Engineering - 3 (ESCAPE-3)*. Suppl. *Computers and Chemical Engineering*, **18**, S675-S680.
- Smith, G.J., and Morton, W. 1988. Dynamic Simulation Using an Equation-Oriented Flowsheeting Package. *Computers and Chemical Engineering*, **12** (5), 469-473.
- Stadtherr, M.A., and Wood, E.S. 1984a. Sparse Matrix Methods for Equation-Based Chemical Process Flowsheeting - I Reordering Phase. *Computers and Chemical Engineering*, **8** (1), 9-18.
- Stadtherr, M.A., and Wood, E.S. 1984b. Sparse Matrix Methods for Equation-Based Chemical Process Flowsheeting - II Numerical Phase. *Computers and Chemical Engineering*, **8** (1), 19-33.

- Stephanopoulos, G., Henning, G., and Leone, H. 1990a. MODEL.LA. A Modeling Language for Process Engineering-I. The Formal Framework. *Computers and Chemical Engineering*, **14** (8), 813-846.
- Stephanopoulos, G., Henning, G., and Leone, H. 1990b. MODEL.LA. A Modeling Language for Process Engineering-II. Multifaceted Modeling of Processing Systems. *Computers and Chemical Engineering*, **14** (8), 847-869.
- Stephanopoulos, G., Johnston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis, M., and Siletti, C. DESIGN-KIT: An Object-oriented Environment for Process Engineering. *Computers and Chemical Engineering*, **11** (6), 655-674.
- Timár, L., Simon, F., Csermely, Z., Siklós, J., Bácksai, S., and Édes, J. 1984. Useful Combination of the Sequential and Simultaneous Modular Strategy in a Flowsheeting Programme. *Computers and Chemical Engineering*, **8** (3/4), 185-194.
- Unger, J., Kröner, A., and Marquardt, W. 1995. Structural Analysis of Differential-Algebraic Equation Systems - Theory and Applications. *Computers and Chemical Engineering*, **19** (8), 867-882.
- Vázquez-Román, R., King, J.M.P., and Bãnares-Alcántara, R. 1996. KBMoSS: A Process Engineering Modelling Support System. In *Proceedings of the European Symposium on Computer-Aided Process Engineering - 6 (ESCAPE-6)*. Suppl. *Computers and Chemical Engineering*, **20**, S309-S314.
- Villadsen, J. 1989. Simulation of Biochemical Reactions. *Computers and Chemical Engineering*, **13** (4/5), 385-395.
- Wayburn, T.L., and Seader, J.D. 1987. Homotopy Continuation Methods for Computer-Aided Process Design. *Computers and Chemical Engineering*, **11** (1), 7-25.
- Westerberg, A.W. 1979. *Process Flowsheeting*. Ch. 4, p105. Cambridge University Press.
- Westerberg, A.W., and Benjamin, D.R. 1985. Thoughts on a Future Equation-Oriented Flowsheeting System. *Computers and Chemical Engineering*, **9** (5), 517-526.
- Williams, T.J. and Otto, R.E. 1960. A Generalized Chemical Processing Model for the Investigation of Computer Control. A.I.E.E. Transactions, **79**, Part 1 (Communications and Electronics), 458-467.
- Ye, N. and McAvoy, T.J. 1995. Optimal Averaging Level Control for the Tennessee Eastman Problem. *Canadian Journal of Chemical Engineering*, **73** (April), 234-240.

# NOMENCLATURE

Due to the complexity of the biochemical process model (ref. Chapter 5, p155-156), the general nomenclature for the thesis and the biochemical process are presented separately.

## General

|                 |   |                   |
|-----------------|---|-------------------|
| $A$             | Area  | $m^2$             |
| $C$             | Valve flow coefficient                          | $kmol/s/Pa^{0.5}$ |
| $c(t)$          | Controller signal                               |                   |
| $C_{pL}$        | Liquid molar specific heat                      | $kJ/kmol/K$       |
| $C_{pV}$        | Vapour molar specific heat                      | $kJ/kmol/K$       |
| $c_s$           | Controller steady-state signal                  |                   |
| $e(t)$          | Controller error signal                         |                   |
| $F$             | Feed stream flowrate <i>or</i> process flowrate | $kmol/s$          |
| $h$             | Liquid height                                   | $m$               |
| $I(t)$          | Integral of controller error signal             | -                 |
| $K_c$           | Controller gain                                 | -                 |
| $K_i$           | Equilibrium constant for component I            |                   |
| $L$             | Liquid stream flowrate                          | $kmol/s$          |
| $M_i$           | Mass holdup of component i                      | $kg$              |
| $M_T$           | Total mass holdup                               | $kg$              |
| $N_i$           | Total molar holdup of component i               | $kmol$            |
| $N_L$           | Total molar holdup of liquid phase              |                   |
| $N_{L,i}$       | Molar holdup in liquid phase of component I     | $kmol$            |
| $N_V$           | Total molar holdup of vapour phase              |                   |
| $N_{V,i}$       | Molar holdup in vapour phase of component I     | $kmol$            |
| $P$             | Pressure  | $Pa$              |
| $R$             | Ratio   |                   |
| $T$             | Temperature                                     | $K$               |
| $V$             | Vapour stream flowrate                          | $kmol/s$          |
| $V_L$           | Liquid phase volume                             | $m^3$             |
| $V_{tot}$       | Total volume                                    | $m^3$             |
| $V_V$           | Vapour phase volume                             | $m^3$             |
| $x$             | Valve position                                  |                   |
| $x_i, y_i, z_i$ | Mass <i>or</i> mole fraction                    |                   |
| $y_m(t)$        | Controller measured value                       | -                 |
| $y_{sp}(t)$     | Controller setpoint                             | -                 |
| $\rho$          | Mass density                                    | $kg/m^3$          |
| $\rho_L$        | Molar density of liquid phase                   | $kmol/m^3$        |
| $\rho_V$        | Molar density of vapour phase                   | $kmol/m^3$        |
| $\tau_I$        | Integral time                                   | $s$               |

## Biochemical Model

|                 |                                      |                              |
|-----------------|--------------------------------------|------------------------------|
| -               | plasmid-free biomass                 |                              |
| +               | plasmid-carrying biomass             |                              |
| $A$             | acetate concentration                | (g acetate) /L               |
| $F_S$           | glucose feed rate                    | (g glucose) / h              |
| $G_p(\mu^+)$    | plasmid concentration in biomass     | (mg plasmid) / (g biomass)   |
| $I$             | IPTG concentration                   | (g IPTG) /L                  |
| $k_a$           | volumetric mass transfer coefficient | $h^{-1}$                     |
| $k_r$           | IPTG recovery rate variable          |                              |
| $K_S$           | saturation constant for glucose      | (g glucose) /L               |
| $k_s$           | IPTG shock rate variable             |                              |
| $\mu$           | specific growth rate                 | $h^{-1}$                     |
| $N$             | agitator rotational speed            | rev /s                       |
| $O_2$           | oxygen concentration                 | (mmol $O_2$ ) /L             |
| $O_2^*$         | saturated oxygen concentration       | (mmol $O_2$ ) /L             |
| $P$             | protein product concentration        | (g protein) /L               |
| $P_F$           | fermentor absolute pressure          | kPa                          |
| $P_M$           | mixing power                         | W                            |
| $P_{O_2}$       | oxygen partial pressure              | kPa                          |
| $p_r$           | probability of plasmid loss          |                              |
| $r_{fa}(\mu)$   | rate of formation of acetate         | (g acetate) / (g biomass) /h |
| $r_{fp}(\mu^+)$ | rate of formation of protein product | (g protein) / (g biomass) /h |
| $R_r$           | IPTG recovery ratio                  |                              |
| $S$             | glucose concentration                | (g glucose) /L               |
| $u_s$           | superficial gas velocity             | m /s                         |
| $V$             | broth volume                         | L                            |
| $X$             | biomass concentration                | (g biomass) /L               |



# APPENDICES

**Appendix A:** General member function descriptions

**Appendix B:** Flash class member functions

**Appendix C:** Tennessee Eastman Unit Operation Models

**Appendix D:** Tennessee Eastman Flowsheet Definition File

**Appendix E:** Recombinant fermentation model parameters

## Appendix A: General Member Function Descriptions

### A.1 System-based classes

#### A.1.1 System Connectivity and Mathematical interface functions

`incorp_main_ss_set(Equation_Set& e)` attaches an executive-level **Equation\_Set** pointer to the low-level **Equation\_Set** object `e`. The **Equation\_Set** object is then available to **System** for analysis. A **Dynamic\_Set** object can also be passed as an argument for steady-state analysis of a dynamic system. The `&` symbol denotes a reference type in C++, which is similar to a pointer (Ellis and Stroustrup 1994), so `e` is then a reference to an **Equation\_Set** object. The function permits different **Equation\_Set** objects to be employed within a **System**.

`incorp_main_dyn_set(Dynamic_Set& d)` has the same function as above, except for dynamic analysis within **System**.

`set_sys(int n)` sets the number of **System**-type objects that the **System** contains or attaches to.

`incorp_sys(System& sys, int n)` attaches a **System**-type object to the  $n^{\text{th}}$  **System** pointer in a **Vector** object named `subsys`. This function and the function above are employed directly in the **Flowsheet** class and for constructing multi-**System** models.

`set_no_inpstrms(int n)` sets the number of **Stream**-type objects entering the **System** object and hence the number of **Input\_Port**-type objects. The function could equally be named `set_no_inp_ports(int n)` but it is more natural to perform connections with **Stream**-type objects.

`own_input_port(Input_Port& p, int n)` attaches an **Input\_Port**-type object to the  $n^{\text{th}}$  **Input\_Port** pointer.

`inp_stream(Stream& strm, int n)` attaches a **Stream**-type object to the **Input\_Port**-type pointed to by the  $n^{\text{th}}$  **Input\_Port** pointer.

`set_no_outstrms(int n)` sets the number of **Stream**-type objects leaving the **System** object and hence the number of **Output\_Port**-type objects.

`own_output_port(Output_Port& p, int n)` attaches an **Output\_Port**-type object to the  $n^{\text{th}}$  **Output\_Port** pointer.

`out_stream(Stream& strm, int n)` attaches a **Stream**-type object to the **Output\_Port**-type pointed to by the  $n^{\text{th}}$  **Output\_Port** pointer.

`set_no_inststrms(int n)` sets the number of internal **Stream**-type objects in a **System** object. This function is not actually required for **Stream**-type objects to be connectors within a **System** object. A **System** object already “owns” **Port**-types as part of the executive structure. Connections are made with **Port**-type objects to **Stream**-type objects and so the connectivity is automatically available.

`incorp_strm(Stream& strm, int n)` attaches a **Stream**-type object to the  $n^{\text{th}}$  **Stream** pointer. This function and the function above are not actually required to define a connected **System**; they are included for the sake of completeness in case other combined **System-Stream** types are developed. One possibility is the creation of a class for modelling long pipe runs where process conditions could result in stratified two-phase flow or significant pressure drop.

### A.1.2 System Analysis

`setup()` drives the mapping functions of the **Port**-type objects in a **System** object.

The function must be redefined for each new modelling class.

`ss_analyse()` performs depth-first connection and steady-state analysis of **System**-types.

`ss_build()` performs depth-first connection and builds **Vector** objects containing **Equation** and **Variable** pointers for steady-state solution.

`reset_ss_eqns()` resets the **System** object's steady-state **Equation\_Set** object so it can be reanalysed and recollected. Required for steady-state initialisation of a dynamic simulation.

`dyn_analyse()` performs depth-first connection and dynamic analysis of **System**-types.

`dyn_build()` performs depth first connection and builds **Vector** objects containing **Equation**, **Variable** and **Derivative** pointers for dynamic solution.

`reset_dyn_eqns()` resets the **Dynamic\_Set** object so it can be reanalysed and recollected.

### A.1.3 Convergence Block class interfaces

`set_no_tear_streams(int n)` sets the number of **Process\_Stream** objects to be torn.

`tear(Process_Stream& strm, int n)` assigns a **Process\_Stream** object to be the  $n^{\text{th}}$  torn stream.

`tear_and_reassign()` tears and reassigns **Variable** objects prior to solution.

`seq_solver(char s[])` specifies the numerical method to be employed in the sequential-modular iterations. Valid arguments are “NEWT”, “BROY”, “MARQ”, “DIRS”, “WEGS” or “NONE”. “DIRS” and “WEGS” correspond to Direct Substitution and Wegstein methods. “NONE” switches off sequential-modular solution.

`sim_solver(char s[])` specifies the numerical method to be employed in the simultaneous/equation-oriented flowsheet iterations. Valid arguments are “NEWT”, “BROY”, “MARQ” or “NONE”.

`solve()` solves the **Flowsheet** object.

## A.2 Port-based classes

### A.2.1 Port, Input Port and Output Port class interface functions

`get_vars(Vector<Variable*>& v)` is a member of the **Port** class. It is the interrogation function described above. The **Vector** object `v` must be allocated (i.e. with a `build(i, j)` call) before it is passed to the function. The function assigns the pointers in a low-level **Port** object to the vector elements. This function is usually only used for debugging.

`map()` is a virtual member of the **Port** class. It connects input and output **Variable** objects to each other. Specific `map()` functions are discussed later in the text.

`set_sink(Stream& str)` is a member of the **Output\_Port** class. It sets the **Port**'s sink **Stream** pointer to `str`. This function is automatically called by more specific **Output\_Port** classes.

`set_source(Stream& str)` is a member of the **Input\_Port** class. It sets the port's source **Stream** pointer to `str`. This function is automatically called by more specific **Input\_Port** classes.

`check_inputs(int n)` is a member of the **Input\_Port** class. It is used to remove input variables from an equation set analysis. Valid arguments are ON or OFF. **Input\_Port**-types default with a call to `check_inputs(OFF)`.

### A.2.2 Process\_Output\_Port and Process\_Input\_Port class interface functions

#### Process\_Output\_Port:

`set_temp_outlet(Variable& v)` sets a **Variable** object for the temperature of the stream leaving through the port. (Normally the vessel temperature).

`set_press_outlet(Variable& v)` sets a **Variable** object for the pressure of the stream leaving through the port. (Normally the vessel pressure).

`set_tot_flow(Variable& v)` sets the total outlet flow **Variable** object.

`set_fracs(Vector<Variable>& v)` sets the composition **Vector** object of the stream leaving through the port.

`get_vars(Vector<Variable*>& v)` obtains a list of the pointer attributes in the port. The elements of the **Vector** `v` are assigned in order of: composition, total flow, output temperature, output pressure, downstream temperature and downstream pressure. The user must know in advance how many components are in a stream and build `v` appropriately.

These member functions above are usually sufficient for general-purpose modelling. Member functions for bi-directional information flow are described below:

`get_temp_output()` returns a pointer to the output temperature **Variable** object of the stream.

`get_press_output()` returns a pointer to the output pressure **Variable** object of the stream.

`get_temp_sink()` returns a pointer to the temperature **Variable** object of the downstream port or vessel.

`get_press_sink()` returns a pointer to the temperature **Variable** object of the downstream port or vessel.

### **Process Input Port:**

`set_temp_owner(Variable& v)` assigns a **Variable** object for the temperature of the environment the stream is entering. (Normally the vessel temperature).

`set_press_owner(Variable& v)` assigns a **Variable** object for the pressure of the environment the stream is entering. (Normally the vessel pressure).

`set_temp_inlet(Variable& v)` assigns a **Variable** object for the temperature of the entering stream. The inlet temperature connection is set by the upstream port.

`set_press_inlet(Variable& v)` assigns a **Variable** object for the pressure of the entering stream. The inlet pressure connection is set by the upstream port.

`set_tot_flow(Variable& v)` assigns the total inlet flow **Variable** object .

`set_fracs(Vector<Variable>& v)` assigns the composition **Vector** of the inlet stream.

`get_vars (Vector<Variable*>& v)` obtains a list of the pointer attributes in the port. The elements of the **Vector** `v` are assigned in order of: composition, total flow, input temperature, input pressure, owner temperature and owner pressure. The user must know in advance how many components are in a stream and build `v` appropriately.

Member functions for bi-directional information flow are described below:

`get_temp_input ()` returns a pointer to the input temperature **Variable** object of the stream.

`get_press_input ()` returns a pointer to the input pressure **Variable** object of the stream.

`get_temp_owner ()` returns a pointer to the temperature **Variable** object of the owner vessel.

`get_press_owner ()` returns a pointer to the temperature **Variable** object of the owner vessel.

### A.2.3 Signal Input Port and Signal Output Port class interface functions

These functions apply to both classes.

`set_signal_var (Variable& v)` assigns the signal **Variable** object of the port.

`get_vars (Vector<Variable*>& v)`. The **Vector** object contains a single element, which is set to the port signal.

### A.2.4 Energy Input Port and Energy Output Port class interface functions



These functions apply to both classes.

`set_energy_var(Variable& v)` assigns the energy **Variable** object of the port.

`get_vars(Vector<Variable*>& v)` . The **Vector** object contains a single element, which is set to the energy **Variable**.

### A.3 Stream classes

#### A.3.1 Stream class interface functions

`get_source()` returns the address of the source **Port** object.

`get_sink()` returns the address of the sink **Port** object.

### A.4 Variable-based classes

#### A.4.1 Variable class interface functions

`operator()` is an overloaded operator. It returns the double precision value of the **Variable**, or the value of the **Variable** object it is connected to. See the example below.

`operator = (double)` is an overloaded = operator that assigns a double precision value to the **Variable** object or the object it is connected to.

An example of usage is :

```
Variable v1,v2,v3;  
v1 = v2() + v3();
```

The = operator is deliberately restricted to operating only on double precision values to enforce the use of a consistent method for specifying mathematical operations and to reduce the amount of recoding required to perform numerical calculations with the **Variable** class.

`set_val(double d)` performs the same function as the = operator.

`connect_to(Variable& v)` assigns the connection of the **Variable** object.

`connect_to(Variable* v)` assigns the connection of the **Variable** object.

`lower(double d)` sets the lower bound of the **Variable** object.

`lower()` returns the lower bound of the **Variable** object.

`upper(double d)` sets the upper bound of the **Variable** object.

`upper()` returns the upper bound of the **Variable** object.

`get_type()` returns the type of the **Variable** object, 'c' for a constant or parameter and 'v' for a potential solution variable.

`constant()` sets the type of the **Variable** object to 'c' (see `get_type()` above).

`var()` sets the type of the **Variable** object to 'v' (see `get_type()` above).

`check()` returns true if the **Variable** is to be checked as part of an **Equation\_Set** analysis or false if it is not.

`check(int n)` sets the analysis status of the **Variable** object. If called as `check(ON)` the **Variable** object will be included in an **Equation\_Set** analysis, or excluded if called as `check(OFF)`.

`get_connection()` returns the immediate connection of the **Variable** object if it has one.

`get_end_connection()` returns the final connection of the **Variable** object. It finds the end of a linked list of connected **Variable** objects (extension of `get_connection()` above).

#### A.4.2 Derivative class interface functions

`set_state(Variable& v)` sets the state **Variable** object of the derivative.

`operator =`. The **Derivative** class cannot employ the `=` operator of the **Variable** class; it must be redefined even though it performs the same function. This is a C++ restriction.

`get_type()` returns the type of the state **Variable** object associated with the **Derivative** object.

#### A.4.3 Equation class interface functions

`set_no_x(int n)` assigns the number of **Variables** that affect the **Equation**.

`include(Variable& v)` includes a **Variable** in the list.

`set_derivative(Derivative& d)` assigns the **Derivative** object of the **Equation**.

`set_exp_var(Variable& v)` assigns the explicit **Variable** for the **Equation**.

#### A.4.4 Equation\_Set and Dynamic\_Set class interface functions

##### Equation\_Set:

`incorp_eqns(Vector<Equation>& e)` attaches a **Vector** of **Equation** objects to the **Equation\_Set** object.

`set_no_subsets(int n)` sets the number of other **Equation\_Set** objects that an **Equation\_Set** object contains or attaches to.

`incorp_set(Equation_Set &e, int n)` attaches an **Equation\_Set** object to the  $n^{\text{th}}$  subset pointer.

`incorp_set(Equation_Set &e)` is the same as above except the **Equation\_Set** object is attached to the next available subset pointer.

`get_no_vars()` returns the number of solution **Variable** objects in the **Equation\_Set**.

`get_no_eqns()` returns the number of solution **Equation** objects in the **Equation\_Set**.

`check()` returns true if the **Equation\_Set** is to be analysed or false if it is not.

`check(int n)` sets the analysis status of the **Equation\_Set**. If called as `check(ON)` the **Equation\_Set** will be analysed, or excluded if called as `check(OFF)`.

##### Dynamic\_Set:

`set_indep(Variable& v)` sets the independent **Variable** object for the **Dynamic\_Set**.

`set_no_ae_sets(int n)` calls the `set_no_subsets(int n)` function above. This function is implemented for clarity in specifying whether a subset is algebraic or dynamic.

`incorp_ae_set(Equation_Set& e, int n)` calls the `incorp_set(Equation_Set &e, int n)` function above.

`incorp_ae_set(Equation_Set& e)` calls the `incorp_set(Equation_Set &e)` function above.

`set_no_dyn_subsets(Dynamic_Set& d)` sets the number of **Dynamic\_Sets** that the **Dynamic\_Set** contains or attaches to.

`incorp_dyn_set(Dynamic_Set &d, int n)` attaches a **Dynamic\_Set** to the  $n^{\text{th}}$  `dyn_subset` pointer.

`incorp_dyn_set(Dynamic_Set &e)` is the same as above except the **Dynamic\_Set** is attached to the next available `dyn_subset` pointer.

`get_no_ae_vars()` returns the number of algebraic solution **Variable** objects in the **Dynamic\_Set**.

`get_no_ae_eqns()` returns the number of algebraic solution **Equation** objects in the **Dynamic\_Set**.

`get_no_dyn_vars()` returns the number of state solution **Variable** objects in the **Dynamic\_Set**.

`get_no_dyn_eqns()` returns the number of state solution **Equation** objects in the **Dynamic\_Set**.

`get_no_derivs()` returns the number of **Derivative** objects in the **Dynamic\_Set**.

## A.5 Physical Property Classes

### A.5.1 Component class interface functions

`get_MW()` returns the molecular weight.

`get_Tc()` returns the critical temperature.

`get_Pc()` returns the critical pressure.

`get_Vc()` returns the critical volume.

`get_Tb()` returns the boiling point.

`get_Tf()` returns the freezing point.

`get_w()` returns the acentric factor.

`get_dipm()` returns the dipole moment.

`get_CpL()` returns the liquid specific heat.

`get_CpV()` returns the vapour specific heat.

`get_rhoL()` returns the liquid density.

`get_rhoV()` returns the vapour density.

`get_Hfg()` returns the enthalpy of vapourisation.

### A.5.2 User Component class interface functions

`set_MW(double d)` sets the molecular weight.

`set_Tc(double d)` sets the critical temperature.

`set_Pc(double d)` sets the critical pressure.

`set_Vc(double d)` sets the critical volume.

`set_Tb(double d)` sets the boiling point.

`set_Tf(double d)` sets the freezing point.

`set_w(double d)` sets the acentric factor.

`set_dipm(double d)` sets the dipole moment.

`set_CpL(double d)` sets the liquid specific heat.

`set_CpV(double d)` sets the vapour specific heat.

`set_rhoL(double d)` sets the liquid density.

`set_rhoV(double d)` sets the vapour density.

`set_Hfg(double d)` sets the enthalpy of vapourisation.

### A.5.3 **Component\_Set** class interface functions

`Component_Set(int n)` is a constructor. The argument `n` sets the number of **Components** in the set.

`incorp_comp(Component& c, int n)` attaches a **Component** to the  $n^{\text{th}}$  position in the set.

`set_datafile(istream& datafile)` sets the text file containing the **Component** property data.

`get_properties()` gets the properties for the incorporated **Components** from the `datafile` object above.

#### A.5.4 General Component Mixture class interface functions

`incorp_compset(Component_Set& cs)` attaches a **Component\_Set** object to the mixture.

`incorp_thermo(Thermo& t)` attaches a **Thermo**-type object to the mixture. The **Component\_Set** for the mixture is automatically attached to the **Thermo** object.

`incorp_vle(VLE& v)` attaches a **VLE**-type object to the mixture. The **Component\_Set** for the mixture is automatically attached to the **VLE** object.

`set_mix_frac(Vector<Variable> &v)` sets the mole fractions for the total mixture.

`set_vap_frac(Vector<Variable> &v)` sets the mole fractions for the vapour phase.

`set_liq_frac(Vector<Variable> &v)` sets the mole fractions for the liquid phase.

`set_To(double T)` sets the reference temperature for the mixture.

`set_Po(double P)` sets the reference pressure for the mixture.

`vap_ave_MW()` returns the average molecular weight for the vapour phase.

`liq_ave_MW()` returns the average molecular weight for the liquid phase.

`ave_MW()` returns the average molecular weight for the mixture.



`CpL_molar(double T, double P)` returns the molar specific heat of the liquid phase at temperature T and pressure P.

`CpL_mass(double T, double P)` returns the mass specific heat of the liquid phase at temperature T and pressure P.

`CpV_molar(double T, double P)` returns the molar specific heat of the vapour phase at temperature T and pressure P.

`CpV_mass(double T, double P)` returns the mass specific heat of the vapour phase at temperature T and pressure P.

`rhoL_molar(double T, double P)` returns the molar density of the liquid phase.

`rhoL_mass(double T, double P)` returns the mass density of the liquid phase.

`rhoV_molar(double T, double P)` returns the molar density of the vapour phase.

`rhoV_mass(double T, double P)` returns the mass density of the vapour phase.

`HL_molar(double T, double P)` returns the molar enthalpy of the liquid phase.

`HL_mass(double T, double P)` returns the molar enthalpy of the liquid phase.

`HV_molar(double T, double P)` returns the molar enthalpy of the vapour phase.

HV\_mass(double T, double P) returns the mass enthalpy of the vapour phase.

SL\_molar(double T, double P) returns the molar entropy of the liquid phase.

SL\_mass(double T, double P) returns the mass entropy of the liquid phase.

SV\_molar(double T, double P) returns the molar entropy of the vapour phase.

SV\_mass(double T, double P) returns the mass entropy of the vapour phase.

GL\_molar(double T, double P) returns the molar Gibbs energy of the liquid phase.

GL\_mass(double T, double P) returns the mass Gibbs energy of the liquid phase.

GV\_molar(double T, double P) returns the molar Gibbs energy of the vapour phase.

GV\_mass(double T, double P) returns the mass Gibbs energy of the vapour phase.

AL\_molar(double T, double P) returns the molar Helmholtz energy of the liquid phase.

AL\_mass(double T, double P) returns the mass Helmholtz energy of the liquid phase.

AV\_molar(double T, double P) returns the molar Helmholtz energy of the vapour phase.

`AV_mass(double T, double P)` returns the mass Helmholtz energy of the vapour phase.

`Ki(Vector<double> &K, double T, double P)` returns the equilibrium constants (K values) for the mixture.

#### A.5.5 Ideal\_VLE class interface functions

`set_no_comps(int n)` sets the number of **Components** in the set.

`set_A(double a, int i)` sets constant A for **Component i**.

`set_B(double b, int i)` sets constant B for **Component i**.

`set_C(double c, int i)` sets constant C for **Component i**.

`Pvapi(Vector<double> &Pvp, double T, double P)` returns vapour pressures of the **Components** in the mixture.

`Ki(Vector<double> &K, double T, double P)` returns the equilibrium constants (K values) for the mixture.

#### A.6 Mathtool class interface functions

`solve_NEWT()` solves simultaneous equations using a Newton method.

`solve_BROY()` solves simultaneous equations using a Broyden method.

`solve_MARQ()` solves simultaneous equations using a Marquardt method.

`setup_solve()` passes a **System** object's analysed steady-state **Equation\_Set** to the solvers.

`setup_integ()` passes a **System** object's analysed **Dynamic\_Set** to the solvers.

`BDF_integrate(double start, double stop, double hstart, double hmin, double hmax, int max_steps, double tol)` employs a variable-step variable-order Gear Backward Difference method to perform the dynamic simulation. Differential and algebraic equations are solved simultaneously as one large set. Integrates over the interval from `start` to `stop`, commencing with a step size `hstart`, a minimum step size  $hmin \leq hstart$ , with a maximum step size `hmax`, a maximum number of steps `max_steps` to an integration error tolerance `tol`.

## Appendix B: Flash Class Member Functions

### B.1 Constructor

```
Flash::Flash(int n, int no_comps, double vol, //Constructor
             double diam):Unit(n){

    int i,j;//Counters.
    nc=no_comps;//Initialise unit parameters.
    Vol=vol;
    Area=PI/4.0*diam*diam;
    hmax=Vol/Area;

    K.build(1,nc);//Allocate vector storage.

    x.build(1,nc);
    y.build(1,nc);
    z.build(1,nc);
    N.build(1,nc);
    Nv.build(1,nc);
    Nl.build(1,nc);
    dNdt.build(1,nc);

    de.build(1,nc+1);
    eqbm.build(1,nc);
    mfs.build(1,2);
    cmb.build(1,nc);
    lmb.build(1,nc);
    vmb.build(1,nc);
    vb.build(1,4);

    for(i=1;i<=nc;i++){//Create dynamic equation map.

        de(i).set_derivative(dNdt(i));
        dNdt(i).set_state(N(i));
        de(i).set_no_x(7);
        de(i).include(z(i));
        de(i).include(F);
        de(i).include(x(i));
        de(i).include(N(i));
        de(i).include(L);
        de(i).include(y(i));
        de(i).include(V);
    }
    de(nc+1).set_derivative(dTdt);
    dTdt.set_state(T);
    de(nc+1).set_no_x(3*nc+10);
    for(i=1;i<=nc;i++){
        de(nc+1).include(x(i));
        de(nc+1).include(y(i));
        de(nc+1).include(z(i));
    }
    de(nc+1).include(NV);
    de(nc+1).include(NL);
    de(nc+1).include(Q);
    de(nc+1).include(L);
```

```

de(nc+1).include(V);
de(nc+1).include(F);
de(nc+1).include(T);
de(nc+1).include(P);
de(nc+1).include(Tin);
de(nc+1).include(Pin);

de_set.incorp_eqns(de);
//Finished dynamic equation map.
//Create equilibrium equation map.
for(i=1;i<=nc;i++){
    eqbm(i).set_no_x(2*nc+2);
    for(j=1;j<=nc;j++){
        eqbm(i).include(x(j));
        eqbm(i).include(y(j));
    }
    eqbm(i).include(T);
    eqbm(i).include(P);
}

eqbm_set.incorp_eqns(eqbm);
//Finished equilibrium equation map.

//Create mole fraction summation map.
mfs(1).set_no_x(nc);
for(i=1;i<=nc;i++)mfs(1).include(x(i));

mfs(2).set_no_x(nc);
for(i=1;i<=nc;i++)mfs(2).include(y(i));

mfs_set.incorp_eqns(mfs);
//Finished mole fraction summation map.

//Create component mole balance map.
for(i=1;i<=nc;i++){
    cmb(i).set_no_x(3);
    cmb(i).include(N(i));
    cmb(i).include(Nv(i));
    cmb(i).include(Nl(i));
}

cmb_set.incorp_eqns(cmb);
//Finish component mole balance map.

//Create liquid mole balance map.
for(i=1;i<=nc;i++){
    lmb(i).set_no_x(3);
    lmb(i).include(NL);
    lmb(i).include(x(i));
    lmb(i).include(Nl(i));
}

lmb_set.incorp_eqns(lmb);
//Finished liquid mole balance map.

//Create vapour mole balance map.
for(i=1;i<=nc;i++){
    vmb(i).set_no_x(3);
    vmb(i).include(NV);
}

```

```

        vmb(i).include(y(i));
        vmb(i).include(Nv(i));
    }

vmb_set.incorp_eqns(vmb);
//Finished vapour mole balance map.

//Create volume balance map.
vb(1).set_no_x(2);
vb(1).include(VL);
vb(1).include(VV);

vb(2).set_no_x(nc+2);
for(i=1;i<=nc;i++)vb(2).include(x(i));
vb(2).include(VL);
vb(2).include(NL);

vb(3).set_no_x(nc+2);
for(i=1;i<=nc;i++)vb(3).include(y(i));
vb(3).include(VV);
vb(3).include(NV);

vb(4).set_no_x(2);
vb(4).include(VL);
vb(4).include(h);

vb_set.incorp_eqns(vb);
//Finished volume balance map.

//Create equation set structure.
de_set.set_nc_ae_sets(6);
de_set.incorp_ae_set(eqbm_set,1);
de_set.incorp_ae_set(mfs_set,2);
de_set.incorp_ae_set(cmb_set,3);
de_set.incorp_ae_set(lmb_set,4);
de_set.incorp_ae_set(vmb_set,5);
de_set.incorp_ae_set(vb_set,6);

incorp_main_dyn_set(de_set);
incorp_main_ss_set(de_set);
//Finished equation set structure.

//Allocate interface variables
feed.set_fracs(z);
feed.set_temp_input(Tin);
feed.set_press_input(Pin);
feed.set_temp_owner(T);
feed.set_press_owner(P);

vapour.set_fracs(y);
vapour.set_temp_output(T);
vapour.set_press_output(P);

liquid.set_fracs(x);
liquid.set_temp_output(T);
liquid.set_press_output(P);

level_sig.set_signal_var(h);
press_sig.set_signal_var(P);

```

```

temp_sig.set_signal_var(T);
heat.set_signal_var(Q);
//Finished interface variables.

//Create process structure.
set_no_inpstrms(2);
own_input_port(feed,1);
own_input_port(heat,2);

set_no_outstrms(5);
own_output_port(liquid,1);
own_output_port(vapour,2);
own_output_port(level_sig,3);
own_output_port(temp_sig,4);
own_output_port(press_sig,5);
//Finished process structure.

//Initialise physical properties.
VL_mix.set_no_comps(nc);
VL_mix.set_liq_frac(x);
VL_mix.set_vap_frac(y);
VL_mix.set_mix_frac(z);
//Finished physical properties.

} //END Constructor.

```

## B.2 Port setup

```

void Flash::setup(){//Connects input and outputs.
//Run by flowsheet.

    feed.map();
    heat.map();

    liquid.map();
    vapour.map();
    level_sig.map();
    temp_sig.map();
    press_sig.map();
}

```

## B.3 Connection Functions

```

//Connection functions.
void Flash::feed_in(Stream& str){
    inp_stream(str,1);
}

void Flash::heat_in(Stream& str){
    inp_stream(str,2);
}

void Flash::liquid_out(Stream& str){
    out_stream(str,1);
}

```



```
void Flash::vapour_out(Stream& str){
    out_stream(str,2);
}

void Flash::level_out(Stream& str){
    out_stream(str,3);
}

void Flash::temp_out(Stream& str){
    out_stream(str,4);
}

void Flash::press_out(Stream& str){
    out_stream(str,5);
}
```

## Appendix C: Tennessee Eastman Unit Models

In all models, the subscript index  $i = 1..8$  corresponds to components A...H. Physical property data for the simulations was calculated from pure component properties as presented in the original paper by Downs and Vogel (1993). Nomenclature is presented at the end of the appendix. Models are presented in an equation-oriented form.

### C.1 Mixer Model

The mixer model is based on a trivial gas-phase material and energy balance. The total mixer volume is  $141.6 \text{ m}^3$ .

$$\frac{dN_{Vi}}{dt} - \sum_{j=1}^5 F_j z_{ji} + Vy_i = 0 \quad i = 1..8 \quad (\text{B.1})$$

$$\frac{dT}{dt} - \sum_{j=1}^5 F_j H_{Fj} + VH_V = 0 \quad (\text{B.2})$$

$$y_i - \frac{N_{Vi}}{\sum_{j=1}^8 N_{Vj}} = 0 \quad i = 1..8 \quad (\text{B.3})$$

$$P - \frac{RT \sum_{j=1}^8 N_{Vj}}{V_{mix}} = 0 \quad (\text{B.4})$$

## C.2 Reactor Model

The reactor model is presented in equation-oriented form below. The reaction kinetic expressions are overleaf. The vapour phase contains a partial pressure balance based on contributions from the condensible and incondensable components. The total reactor volume is 36.8 m<sup>3</sup>.

$$\frac{dN_i}{dt} - Fz_i + Vy_i - X_{Ri} = 0 \quad i = 1 \dots 8 \quad (\text{B.5})$$

$$\frac{dT}{dt} - \left( \frac{1}{C_{PL} \sum_{j=1}^8 N_{Lj} + C_{PV} \sum_{j=1}^8 N_{Vj}} \right) (FH_F - VH_V + H_R + Q) = 0 \quad (\text{B.6})$$

$$\frac{dT_w}{dt} - \frac{1}{M_w C_{PW}} (x_Q F_{W_{\max}} C_{PW} (T_{W_{in}} - T_w) - Q) = 0 \quad (\text{B.7})$$

$$Q - UA \frac{V_L}{V_{L_{\max}}} (T - T_w) = 0 \quad (\text{B.8})$$

$$V_L - \frac{\sum_{j=1}^8 N_{Lj}}{\rho_L} = 0 \quad (\text{B.9})$$

$$y_{Ci} - \frac{x_i P_{\text{sat}i}}{P_C} = 0 \quad i = 4 \dots 8 \quad (\text{B.10})$$

$$\sum_{j=1}^3 y_{Uj} - 1 = 0 \quad (\text{B.11})$$

$$\sum_{j=1}^8 y_{Cj} - 1 = 0 \quad (\text{B.12})$$

$$y_i - \left( \frac{P - P_C}{P} \right) y_{Ui} = 0 \quad i = 1 \dots 3 \quad (\text{B.13})$$

$$y_i - \left( \frac{P_C}{P} \right) y_{Ci} = 0 \quad i = 4 \dots 8 \quad (\text{B.14})$$

$$N_{Vi} - \frac{Py_i V_V}{RT} = 0 \quad i = 1 \dots 8 \quad (\text{B.15})$$

$$N_i - N_{Vi} = 0 \quad i = 1 \dots 3 \quad (\text{B.16})$$

$$N_i - N_{Vi} - N_{Li} = 0 \quad i = 4 \dots 8 \quad (\text{B.17})$$

$$x_i - \frac{N_{Li}}{V_L \rho_L} = 0 \quad i = 4 \dots 8 \quad (\text{B.18})$$

$$\mathbf{R}_1 = 0.454 \exp\left(31.5859536 - \frac{40000}{1.987T}\right) \mathbf{p}'_1{}^{1.1544} \mathbf{p}'_3{}^{0.3735} \mathbf{p}'_4 (35.3147V_V) \quad (\text{B.19})$$

$$\mathbf{R}_2 = 0.454 \exp\left(3.00094014 - \frac{60000}{1.987T}\right) \mathbf{p}'_1{}^{1.1544} \mathbf{p}'_3{}^{0.3735} \mathbf{p}'_5 (35.3147V_V) \quad (\text{B.20})$$

$$\mathbf{R}_3 = 0.454 \exp\left(31.5859536 - \frac{40000}{1.987T}\right) \mathbf{p}'_1 \mathbf{p}'_5 (35.3147V_V) \quad (\text{B.21})$$

$$\mathbf{R}_4 = 0.76748834 \mathbf{R}_3 \mathbf{p}'_1 \mathbf{p}'_4 \quad (\text{B.22})$$

$$\mathbf{H}_R = \mathbf{H}_{R_1} \mathbf{R}_1 + \mathbf{H}_{R_2} \mathbf{R}_2 \quad (\text{B.23})$$

$$\mathbf{X}_{R_1} = -\mathbf{R}_1 - \mathbf{R}_2 - \mathbf{R}_3 \quad (\text{B.24})$$

$$\mathbf{X}_{R_2} = -\mathbf{R}_1 - \mathbf{R}_2 \quad (\text{B.25})$$

$$\mathbf{X}_{R_3} = -\mathbf{R}_1 - 1.5\mathbf{R}_4 \quad (\text{B.26})$$

$$\mathbf{X}_{R_4} = -\mathbf{R}_2 - \mathbf{R}_3 \quad (\text{B.27})$$

$$\mathbf{X}_{R_5} = \mathbf{R}_3 + \mathbf{R}_4 \quad (\text{B.28})$$

$$\mathbf{X}_{R_6} = \mathbf{R}_1 \quad (\text{B.29})$$

$$\mathbf{X}_{R_7} = \mathbf{R}_2 \quad (\text{B.30})$$

### C.3 Separator Model

The separator model is similar to the reactor model, with addition of a liquid product stream and the omission of the reaction kinetics. The total separator volume is 99.1 m<sup>3</sup>.

$$\frac{dN_i}{dt} - Fz_i + Vy_i - Lx_i = 0 \quad i = 1...8 \quad (\text{B.31})$$

$$\frac{dT}{dt} - \left( \frac{1}{C_{PL} \sum_{j=1}^8 N_{Lj} + C_{PV} \sum_{j=1}^8 N_{Vj}} \right) (FH_F - VH_V - LH_L + Q) = 0 \quad (\text{B.32})$$

$$\frac{dT_W}{dt} - \frac{1}{M_W C_{PW}} (x_Q F_{W \max} C_{PW} (T_{W in} - T_W) - Q) = 0 \quad (\text{B.33})$$

$$Q - UA(T - T_W) = 0 \quad (\text{B.34})$$

$$V_L - \frac{\sum_{j=1}^8 N_{Lj}}{\rho_L} = 0 \quad (\text{B.35})$$

$$y_{Ci} - \frac{x_i P_{sati}}{P_C} = 0 \quad i = 4...8 \quad (\text{B.36})$$

$$\sum_{j=1}^3 y_{Uj} - 1 = 0 \quad (\text{B.37})$$

$$\sum_{j=1}^8 y_{Cj} - 1 = 0 \quad (\text{B.38})$$

$$y_i - \left( \frac{P - P_C}{P} \right) y_{Ui} = 0 \quad i = 1...3 \quad (\text{B.39})$$

$$y_i - \left( \frac{P_C}{P} \right) y_{Ci} = 0 \quad i = 4...8 \quad (\text{B.40})$$

$$N_{Vi} - \frac{Py_i V_V}{RT} = 0 \quad i = 1...8 \quad (\text{B.41})$$

$$N_i - N_{Vi} = 0 \quad i = 1...3 \quad (\text{B.42})$$

$$N_i - N_{Vi} - N_{Li} = 0 \quad i = 4...8 \quad (\text{B.43})$$

$$x_i - \frac{N_{Li}}{V_L \rho_L} = 0 \quad i = 4...8 \quad (\text{B.44})$$

## C.4 Stripper Model

The Tennessee Eastman Fortran code contains an unusual model for the stripping column. The model form is not completely clear from the code, but the model appears to assume that vapour phase holdup equilibrium is instantaneous. The stripper vapour volume is neglected in the code. The split between vapour and liquid is calculated from a temperature and feed-flow dependent recovery term. The specific heat of the vapour phase is neglected. The heat duty is regulated by a simple linear-dependence equation. All components are condensible in this model for the purposes of the liquid product stream.

$$\frac{dN_{Li}}{dt} - (1 - \phi_i)(F_1 z_{1i} + F_2 z_{2i}) + Lx_i = 0 \quad i = 1 \dots 8 \quad (\text{B.45})$$

$$\frac{dT}{dt} - \frac{1}{C_{PL} \sum_{j=4}^8 N_{Lj}} (F_1 H_{F1} + F_2 H_{F2} - V H_V - L H_L + Q) = 0 \quad (\text{B.46})$$

$$\phi_i (F_1 z_{1i} + F_2 z_{2i}) - V y_i = 0 \quad i = 1 \dots 8 \quad (\text{B.47})$$

$$\sum_{j=1}^8 y_j - 1 = 0 \quad (\text{B.48})$$

$$\frac{Q}{Q_{\max}} - x_Q = 0 \quad (\text{B.49})$$

$$x_i - \frac{N_{Li}}{\sum_{j=1}^8 N_{Lj}} = 0 \quad i = 1 \dots 8 \quad (\text{B.50})$$

$$V_L - \frac{\sum_{j=1}^8 N_{Lj}}{\rho_L} = 0 \quad (\text{B.51})$$

The recoveries are calculated from the following expressions:

$$T_{fact} = \frac{363.744}{177 - (T - 273.15)} - 2.22579488 \quad (\text{B.52})$$

$$R_{fact} = \frac{F_2}{F_1} T_{fact} \quad (\text{B.53})$$

$$\phi_1 = 0.995 \quad (\text{B.54})$$

$$\phi_2 = 0.991 \quad (\text{B.55})$$

$$\phi_3 = 0.990 \quad (\text{B.56})$$

$$\phi_i = \frac{s_i R_{fact}}{1 + s_i R_{fact}} \quad i = 4 \dots 8 \quad (\text{B.57})$$

## C.5 Nomenclature

All pressures are absolute. The majority of the constants have been converted from values in the Fortran code.

| Symbol     |   | Units          |
|------------|---|----------------|
| $C_{PL}$   | molar specific heat of liquid phase                     | kJ/kmol/K      |
| $C_{PV}$   | molar specific heat of vapour phase                     | kJ/kmol/K      |
| $F$        | flowrate of feed stream                                 | kmol/h         |
| $F_j$      | flowrate of feed stream j                               | kmol/h         |
| $F_j$      | flowrate of feed stream j                               | kmol/h         |
| $H_F$      | specific enthalpy of feed stream                        | kJ/kmol        |
| $H_{Fj}$   | specific enthalpy of feed stream j                      | kJ/kmol        |
| $H_L$      | specific enthalpy of liquid stream                      | kJ/kmol        |
| $H_R$      | heat production rate in reactor                         | kJ/h           |
| $H_V$      | enthalpy of vapour stream                               | kJ/kmol        |
| $N_i$      | total holdup of species i                               | kmol           |
| $N_{Li}$   | liquid phase holdup of species i                        | kmol           |
| $N_{Vi}$   | vapour phase holdup of species i                        | kmol           |
| $P$        | vessel pressure   | kPa            |
| $p'_i$     | partial pressure of component i                         | mm Hg          |
| $P_C$      | total vapour partial pressure of condensible components | kPa            |
| $P_{sati}$ | saturated vapour pressure of component i                | kPa            |
| $Q$        | cooling or heating duty                                 | kJ/h           |
| $Rfact$    | stripper temperature and feedrate recovery factor       |                |
| $R_i$      | rate of reaction i                                      | kmol/h         |
| $T$        | vessel temperature                                      | K              |
| $Tfact$    | stripper temperature factor                             |                |
| $T_W$      | cooling water temperature                               | K              |
| $V$        | vapour flowrate   | kmol/h         |
| $V_L$      | liquid phase volume                                     | m <sup>3</sup> |
| $V_V$      | vapour phase volume                                     | m <sup>3</sup> |
| $x_i$      | mole fraction species i, liquid phase                   |                |
| $x_Q$      | position of cooling/heating supply valve                | 0 - 1          |
| $X_{Ri}$   | production rate of species i in reactor                 | kmol/h         |
| $y_{Ci}$   | mole fraction condensible species i, vapour phase       |                |
| $y_i$      | mole fraction species in vapour phase                   |                |
| $y_{Ui}$   | mole fraction uncondensable species i, vapour phase     |                |
| $z_i$      | mole fraction of component i in feed                    |                |
| $z_{ji}$   | mole fraction of component i in feed stream j           |                |

| Constant   |   | Value   | Units                        |
|------------|---|---|------------------------------|
| $R$        | gas constant  | 8.3144  | $\text{m}^3\text{kPa/kmol/}$ |
| $UA$       | heat transfer coefficient                               | $1.0679 \times 10^6$ (react)<br>$7.0000 \times 10^5$ (sep)                            | $\text{kJ/h/K}$              |
| $H_{R1}$   | specific heat production of reaction 1                  | 150000  | $\text{kJ/kmol}$             |
| $H_{R2}$   | specific heat production of reaction 2                  | 110000  | $\text{kJ/kmol}$             |
| $Q_{max}$  | maximum heating duty available (stripper)               | $1 \times 10^6$   | $\text{kJ/h}$                |
| $F_{wmax}$ | maximum cooling water flowrate available                | 227100 (react)<br>272000 (sep)  | $\text{kg/hr}$               |
| $C_{pw}$   | specific heat of water                                  | 4.186   | $\text{kJ/kg/K}$             |
| $s_i$      | stripper condensible component vapour recovery constant | $s_4 = 8.5010$<br>$s_5 = 11.402$<br>$s_6 = 11.795$<br>$s_7 = 0.048$<br>$s_8 = 0.0242$ |                              |
| $M_w$      | mass holdup of water in cooling coil                    | 5000 (react)<br>7500 (sep)  | $\text{kg}$                  |
| $V_{Lmax}$ | maximum liquid volume in reactor                        | 21.0  | $\text{m}^3$                 |
| $T_{win}$  | cooling water supply temperature                        | 308.15  | $\text{K}$                   |

| Greek symbols | Description                       | Units             |
|---------------|-----------------------------------|-------------------|
| $\rho_L$      | liquid molar density              | $\text{kmol/m}^3$ |
| $\phi_i$      | stripper vapour recovery fraction |                   |



## Appendix D: Tennessee Eastman Flowsheet Definition

A flowsheet example for the definition of the Tennessee Eastman problem with the control system illustrated in Figure 5.5 is presented below. The flowsheet specifies a setpoint change to the reactor pressure.

```
#include "c:\eastman\tenneast.hpp"//header file for the problem

void main(void){//start program

    //declare components, streams, units etc
    //all objects are declared static to avoid overflowing
    //the stack in memory

    static User_Component TennA,TennB,TennC,TennD;
    static User_Component TennE,TennF,TennG,TennH;

    static Component_Set TennSet (8);
    static Ideal_VLE TennVle(8);
    static Simple_Thermo TennTherm(8);

    #include "c:\\eastman\\teprops.hpp"//include component data file
    //this file contains the C++ code for defining component properties
    //etc and where the components are assigned to the set

    //Most names are self explanatory
    static Flowsheet f(999);
    static Boundary_Node Asource(1,8),Csource(2,8);
    static Boundary_Node Dsource(3,8),Esource(4,8);
    static Boundary_Node prod_sink(5,8),purge_sink(6,8);

    static TE_Mixer mix(7,5);
    static TE_Reactor rx(8);
    static TE_Incon_Flash sep(9);
    static TE_Stripper strip(10);
    static Splitter split(11);
    static Compressor compr(12);
    static Pump pmp(13);

    static Simple_Valve vA(14,8),vC(15,8),vD(16,8);
    static Simple_Valve vE(17,8),prod_valve(18,8);

    static Control_Valve v10(19,8),v9(20,8);

    static Flow_Valve vmr(21,8),vrs(22,8),vrec(23,8);

    static PI_Controller PCrx(24),TCrx(25);
    static PI_Controller TCstrip(26),FCprod(27);
    static P_Controller LCsep(28),LCstrip(29),LCrx(30);
    static P_Controller CCA(31);
    static PI_Controller CCB(32);

    static Ratio_Controller RC(33);

    static TE_Analyser prod_an(34),purge_an(35);
```

```

static Flow_Indicator fiprod(36,8);

static Process_Stream Afeed,Cfeed,Dfeed,Efeed;
static Process_Stream str1,str2,str3,str4;
static Process_Stream str5,str6a,str6b;
static Process_Stream str7a,str7b,str8a,str8b;
static Process_Stream str8c,str9a,str9b,str10a;
static Process_Stream str10b,str10c,str11a,str11b;
static Process_Stream sepout,vsepout;

static Signal_Stream RxPsig,RxVsig,rxTsig,rxQsig;
static Signal_Stream vAsig,vCsig,vDsig,vEsig;
static Signal_Stream Gsig,Hsig,Asig,Bsig;
static Signal_Stream SepVsig,v10sig,v9sig;
static Signal_Stream StripVsig,stripTsig,stripQsig;
static Signal_Stream prodflowsig,prodvalvesig;
static Signal_Stream sepTsig,sepQsig;

//END FLOWSHEET AND UNIT DECLARATION

TennSet.incorp_VLE(TennVle); //attach the property objects to the set
TennSet.incorp_Thermo(TennTherm);

//START PROCESS LAYOUT

Asource.out_strm(Afeed);
Csource.out_strm(Cfeed);
Dsource.out_strm(Dfeed);
Esource.out_strm(Efeed);

vA.inp_strm(Afeed,1);
vA.out_strm(str1,1);

vC.inp_strm(Cfeed,1);
vC.out_strm(str4,1);

vD.inp_strm(Dfeed,1);
vD.out_strm(str2,1);

vE.inp_strm(Efeed,1);
vE.out_strm(str3,1);

mix.inp_strm(str1,1);
mix.inp_strm(str2,2);
mix.inp_strm(str3,3);
mix.inp_strm(str8c,4);
mix.inp_strm(str5,5);
mix.out_strm(str6a,1);

vmr.inp_strm(str6a,1);
vmr.out_strm(str6b,1);

rx.inp_strm(str6b,1);
rx.out_strm(str7a,1);

vrs.inp_strm(str7a,1);
vrs.out_strm(str7b,1);

```

```

sep.inp_strm(str7b,1);
sep.out_strm(sepout,1);
sep.out_strm(str10a,2);

pump.inp_strm(str10a,1);
pump.out_strm(str10b,1);

v10.inp_strm(str10b,1);
v10.out_strm(str10c,1);

strip.inp_strm(str10c,1);
strip.inp_strm(str4,2);

strip.out_strm(str5,1);
strip.out_strm(str11a,2);

prod_valve.inp_strm(str11a,1);
prod_valve.out_strm(str11b,1);

prod_sink.inp_strm(str11b);

split.inp_strm(sepout,1);
split.out_strm(str9a,1);
split.out_strm(str8a,2);

v9.inp_strm(str9a,1);
v9.out_strm(str9b,1);

purge_sink.inp_strm(str9b);

compr.inp_strm(str8a,1);
compr.out_strm(str8b,1);

vrec.inp_strm(str8b,1);
vrec.out_strm(str8c,1);
//END PROCESS LAYOUT

//START CONTROL LAYOUT

//start reactor pressure control
PCrx.signal_out(vCsig);
PCrx.signal_in(RxPsig);
rx.pressure_signal(RxPsig);
vC.signal_stream(vCsig);
//end reactor pressure control

//start reactor temperature control
TCrx.signal_in(rxTsig);
TCrx.signal_out(rxQsig);
rx.temp_signal(rxTsig);
rx.heat_signal(rxQsig);
//end reactor temperature control

//start reactor level control
LCrx.signal_out(vEsig);
LCrx.signal_in(RxVsig);
rx.volume_signal(RxVsig);
vE.signal_stream(vEsig);

```

```

//end reactor level control

//start product ratio control
prod_an.set_stream(str11b);
prod_an.out_strm(Gsig,7);
prod_an.out_strm(Hsig,8);

RC.signal_A_in(Gsig);
RC.signal_B_in(Hsig);
RC.signal_out(vDsig);
vD.signal_stream(vDsig);
//end product ratio control

//start production rate control
fiprod.inp_strm(str11b,1);
fiprod.out_strm(prodflowsig,1);
FCprod.signal_in(prodflowsig);
FCprod.signal_out(prodvalvesig);
prod_valve.signal_stream(prodvalvesig);
//end product rate control

//start purge A & B control
purge_an.set_stream(str9b);
purge_an.out_strm(Asig,1);
purge_an.out_strm(Bsig,2);

CCB.signal_in(Bsig);
CCB.signal_out(v9sig);
v9.signal_stream(v9sig);

CCA.signal_in(Asig);
CCA.signal_out(vAsig);
vA.signal_stream(vAsig);
//end purge control

//start separator level control
sep.volume_signal(SepVsig);
LCsep.signal_in(SepVsig);
LCsep.signal_out(sepQsig);
sep.heat_signal(sepQsig);
//end separator level control

//start stripper level control
strip.volume_signal(StripVsig);
LCstrip.signal_in(StripVsig);
LCstrip.signal_out(v10sig);
v10.signal_stream(v10sig);
//end stripper level control

//start stripper temp control
strip.temp_signal(stripTsig);
TCstrip.signal_in(stripTsig);
TCstrip.signal_out(stripQsig);
strip.heat_signal(stripQsig);
//end stripper temp control

//END CONTROL LAYOUT

f.set_sys(36); //36 systems in the flowsheet

```

```

f.incorp_sys(Asource,1);
f.incorp_sys(Csource,2);
f.incorp_sys(Dsource,3);
f.incorp_sys(Esource,4);
f.incorp_sys(vA,5);
f.incorp_sys(vC,6);
f.incorp_sys(vD,7);
f.incorp_sys(vE,8);
f.incorp_sys(mix,9);
f.incorp_sys(vmr,10);
f.incorp_sys(rx,11);
f.incorp_sys(vrs,12);
f.incorp_sys(sep,13);
f.incorp_sys(pump,14);
f.incorp_sys(vl0,15);
f.incorp_sys(strip,16);
f.incorp_sys(prod_valve,17);
f.incorp_sys(prod_sink,18);
f.incorp_sys(split,19);
f.incorp_sys(compr,20);
f.incorp_sys(v9,21);
f.incorp_sys(vrec,22);
f.incorp_sys(LCrX,23);
f.incorp_sys(PCrX,24);
f.incorp_sys(purge_sink,25);
f.incorp_sys(purge_an,26);
f.incorp_sys(prod_an,27);
f.incorp_sys(TCstrip,28);
f.incorp_sys(LCstrip,29);
f.incorp_sys(CCA,30);
f.incorp_sys(fiprod,31);
f.incorp_sys(FCprod,32);
f.incorp_sys(RC,33);
f.incorp_sys(CCB,34);
f.incorp_sys(LCsep,35);
f.incorp_sys(TCrX,36);

f.setup();//set up all the subsystems

//Attach the set of components to each unit
mix.set_comps(TennSet);
rx.set_comps(TennSet);
vA.set_comps(TennSet);
vC.set_comps(TennSet);
vD.set_comps(TennSet);
vE.set_comps(TennSet);
vmr.set_comps(TennSet);
vrs.set_comps(TennSet);
vl0.set_comps(TennSet);
v9.set_comps(TennSet);
prod_valve.set_comps(TennSet);
vrec.set_comps(TennSet);
sep.set_comps(TennSet);
strip.set_comps(TennSet);
fiprod.set_comps(TennSet);
RC.set_comps(TennSet);

//set controller parameters.  Ti is in hours.

```

```

PCrx.set_Kc(0.00125); //output fraction/kPa
PCrx.set_Ti(1.25);

TCrx.set_Kc(-0.03125); //output fraction/K
TCrx.set_Ti(20.0/60.0);

CCA.set_Kc(1.0); //output fraction/mole fraction

LCrx.set_Kc(0.04); //output fraction/m^3

LCsep.set_Kc(0.04); //output fraction/m^3

RC.set_Kc(0.3); //output fraction/ratio fraction
RC.set_Ti(1.0);

LCstrip.set_Kc(0.1); //output fraction/m^3

TCstrip.set_Kc(0.02); //output fraction/K
TCstrip.set_Ti(1.0/6.0);

FCprod.set_Kc(0.15); //output fraction/volume flow fraction
FCprod.set_Ti(5E-3);

CCB.set_Kc(-12.0); //output fraction/mole fraction
CCB.set_Ti(12.0);

fiprod.volume();
fiprod.set_Fmax(49.10); //maximum volume flow, m^3 /hr

prod_an.set_freq(0.25); //sampling frequency and deadtime
purge_an.set_freq(0.1);

#include "c:\\eastman\\ss_est.txt" ); //include file with steady-state
//initial estimates
//open output files for desired units
mix.open_output_file("c:\\eastman\\mix.txt");
rx.open_output_file("c:\\eastman\\rx.txt");
sep.open_output_file("c:\\eastman\\sep.txt");
strip.open_output_file("c:\\eastman\\strip.txt");
PCrx.open_output_file("c:\\eastman\\PCrx.txt");
LCrx.open_output_file("c:\\eastman\\LCrx.txt");
TCrx.open_output_file("c:\\eastman\\TCrx.txt" );
TCstrip.open_output_file("c:\\eastman\\TCstrip.txt");
LCsep.open_output_file("c:\\eastman\\LCsep.txt");
LCstrip.open_output_file("c:\\eastman\\LCstrip.txt");
CC.open_output_file("c:\\eastman\\CC.txt");
CCA.open_output_file("c:\\eastman\\CCA.txt");
RC.open_output_file("c:\\eastman\\RC.txt");
FCprod.open_output_file("c:\\eastman\\fcprod.txt");

//analyse and construct equation set
f.ss_analyse();
f.ss_build();

f.setup_solve(); //send equation set to solver
f.solve_NEWT(); //solve system

//output ss solution to file
mix.ss_output();

```

```

rx.ss_output();
sep.ss_output();
strip.ss_output();
PCrx.ss_output();
LCrx.ss_output();
TCrx.ss_output();
TCsep.ss_output();
TCrxw.ss_output();
TCstrip.ss_output();
LCsep.ss_output();
LCstrip.ss_output();
CC.ss_output();
CCA.ss_output();
RC.ss_output();
FCprod.ss_output();

f.reset_ss_eqns();//reset equations

#include "c:\\eastman\\dyn_specs.txt" );//include file with dynamic
//specifications

//analyse and construct equation set
f.dyn_analyse();
f.dyn_build();

f.setup_integ();//send equations to solver

PCrx.set_sp(2746.0);//change setpoint

//integrate. Parameters are :
//begin time, end time, initial step, minimum step,
//maximum step (null in this case), maximum no. of steps,
//and tolerance
f.BDF_integrate(0.0,24.0,0.0005,1E-8,0.0,150000,1E-4);
}

```

## Appendix E: Fermentation Model Parameters

The parameters for the fermentation model, equations (5.6) - (5.25) are:

|                          |  |           |                                |
|--------------------------|--|-----------|--------------------------------|
| $\alpha$                 | $k_L a$ correlation parameter                            | 0.3       |                                |
| $A_{\max}$               | growth-limiting acetate concentration                    | 15.0      | (g acetate) /L                 |
| $\beta$                  | $k_L a$ correlation parameter                            | 0.7       |                                |
| $C_A$                    | IPTG recovery ratio constant                             | 0.22      | (g IPTG) /L                    |
| $d_s$                    | agitator diameter  | 0.12      | m                              |
| $f_I$                    | IPTG protein production constant                         | 0.0005    | (g IPTG) /L                    |
| $k$                      | $k_L a$ correlation parameter                            | 0.0018    | $m^{2.7} / s^{0.7} / W^{0.7}$  |
| $K_{\theta \varepsilon}$ | yield of protein product from plasmid                    | 41.8      | (mg protein) / (mg plasmid)    |
| $k_I$                    | IPTG shock/recovery rate constant                        | 0.09      | $h^{-1}$                       |
| $K_h$                    | plasmid loss probability equation saturation coefficient | 0.132     |                                |
| $K_I$                    | IPTG saturation constant                                 | 0.034     | (g IPTG) /L                    |
| $K_{O_2}$                | saturation constant for oxygen                           | 0.0027    | (mmol $O_2$ ) /L               |
| $\lambda$                | plasmid loss probability equation parameter              | 0.0015    |                                |
| $\mu_{\max}^-$           | maximum specific growth rate, wild cells                 | 0.75      | $h^{-1}$                       |
| $\mu_{\max}^+$           | maximum specific growth rate, recombinant cells          | 0.70      | $h^{-1}$                       |
| $m_{XO_2}$               | biomass oxygen maintenance coefficient                   | 0.481     | (mmol $O_2$ ) / (g biomass) /h |
| $m_{XS}$                 | biomass glucose maintenance coefficient                  | 0.054     | (g glucose) / (g biomass) /h   |
| $n$                      | plasmid loss probability equation exponent               | 1.78      |                                |
| $N_P$                    | agitator power number                                    | 5.2       |                                |
| $P_{H_2O}$               | vapour pressure of water                                 | 6.233     | kPa @ 37°C                     |
| $P_{\max}$               | growth-limiting protein product concentration            | 1.0       | (g protein) /L                 |
| $\rho$                   | broth density  | 1100      | g /L                           |
| $\rho_S$                 | glucose feed density                                     | 100 - 500 | (g glucose) / (L feed)         |
| $V^0$                    | maximum plasmid replication rate                         | 0.215     | (mg plasmid) / (g biomass) /h  |
| $Y_{AS}$                 | yield of acetate from glucose                            | 0.23      | (g acetate) / (g glucose)      |
| $Y_{PS}$                 | yield of protein product from glucose                    | 0.2       | (g protein) / (g glucose)      |
| $Y_{XO_2}$               | yield of biomass from oxygen                             | 0.03855   | (g biomass) / (mmol $O_2$ )    |
| $Y_{XS}$                 | yield of biomass from glucose                            | 0.5       | (g biomass) / (g glucose)      |