Modelling Environments for Large Scale Process System Problems

David Riach Mitchell

Doctor of Philosophy
The University of Edinburgh
2000

Declaration

The work described in this thesis is the original work of the author and was carried out without the assistance of others, except where explicit credit is given in the text. It has not been submitted, in whole or in part for any other degree at any university.

Acknowledgements

I would like to thank Dr. Bill Morton for his supervision and the support he has provided throughout the project, the various Computing Officers (Neil, Geoff and Colin) who have kept the system going and all the others who have helped me during my time here.

Abstract

This thesis presents a novel modelling environment for large scale process systems problems. Traditional modelling environments attempt to provide maximal functionality within a fixed modelling language. The intention of such systems is to provide the user with a complete package that requires no further development or coding on their part. This approach limits the user to the functionality provided within the package but requires little or no programming experience on the part of the user.

It is argued that for truly novel and complex model development the user must be capable of fully tailoring the environment to their requirements. The environment developed, JFMS (Java based Flexible Modelling System) consists of an object orientated model definition language and a three tier architecture comprising:

- Model building routines;
- Core data structures;
- Application and Method server.

The environment provides sufficient capability for the user to describe the model in terms of a variable set and a set of methods with which to manipulate the variables. Many of these methods will describe equations but there is no restriction limiting methods to representing equations. These methods can act as agents, linking the modelling environment to external systems such as physical property databanks and non-JFMS format models.

Separating the description of the model from its processing allows the complexities to be dealt with in a full programming language (external functions are written in Fortran90 or C). The behaviour of the system is tailored by the user, the modelling environment existing solely to store the model structure and provide the interface layer between the external systems.

Contents

List of Figures

Chapter 1

Introduction

At this time, computers are commonly used in almost all areas of life; washing machines are 'intelligent', vehicle engines are controlled by on-board processors, offices are largely electronic and the Internet paradigm of distributed computing is becoming all pervasive. The area of computing of interest to this project is the modelling, simulation and optimisation field. Given the increase in processing power and reducing memory costs the models inevitably become larger and more complex. What should a modelling environment provide in order to be useful in such a dynamic domain?

1.1 Background

This work has been performed as part of the EPSRC funded Large Scale Optimisation Group (LSOG). This project was started in October 1996 as a collaboration between Edinburgh and Dundee Universities, involving staff from the Maths and Chemical Engineering departments. Many of the staff involved were previously working within the ECOSSE consortium and this project can be viewed as a continuation of the equation based modelling and solution method research performed under that project. The group has three main areas of interest:

- Modelling environments for large process models;

- Novel optimisation and solution methods;
- Pre-processing steps to aid convergence.

Given the background of the staff involved in the project, the models examined have inevitably had a chemical engineering basis. The concepts, methods and applications developed by the group are however equally applicable in the wider numerical modelling domain. This has been deliberate as a truly flexible and adaptable system will need to be capable of representing or manipulating a wide range of modelling problems. The assumption has been made that the domain will change over time and therefore simplifications that are currently valid would inevitably prove restrictive in the longer term. A concerted effort has been made to ensure that methods developed are truly generic.

1.2 Aim

The aim of this thesis is to present the work carried out within two of the above areas, in particular in the modelling environments and pre-processing areas.

1.3 Objectives

The main objective of this project was to provide the necessary modelling support to LSOG. This required identification of an equation based modelling tool sufficiently extensible in order to not restrict the applications and methods being developed elsewhere in the group. As, at the time, it was not possible to find such an application it was decided to produce an in-house package to support the modelling work.

A secondary objective was to review the current pre-processing methods available to assist in creation of good initial values for the variable set. If necessary, new methods were to be developed.

1.4 Scope

Initially, the project was constrained to providing a modelling environment for use within the group. This implied a user type familiar with the concepts of equation based modelling, computer literate and used to command line driven applications. As the project developed it became clear that the application was useful for a wider range of users and, in particular, those at an undergraduate level. After feedback from the first undergraduate users, the project scope widened to produce an application capable of supporting complex development and modelling by expert users while remaining useful for new users, inexperienced in the subject area.

1.5 Computer Based Modelling

Computer modelling is becoming increasingly common in a highly diverse range of disciplines, including vehicle, electronic, structural and process engineering. Typical applications include the design, test and evaluation and simulation of the complex structures and processes found in these domains. Modelling such systems on computers provides a rapid and inexpensive method of satisfying requirements traditionally met by the construction of multiple prototypes or duplicate systems. These requirements include:

- System design;
- System optimisation;
- System simulation.

These models provide a numerical representation of the state and behaviour of the system in question. The models can be classified by intent into one of two classes; simulation or optimisation. These models can in turn be either steady state, where the system state does not change with time, or dynamic, allowing the user to investigate the behaviour of the system over time.

Simulation uses models of the process to investigate the effects of different operating conditions on the real life process. Two types of simulation exist, steady state and dynamic, each of which have different uses. Steady state models are used to study how the process operates in a given design state. Such models are typically used to validate designs, ensuring that the proposed design can achieve the required outputs, or to produce cost estimates for budget purposes. In contrast, dynamic models can be used to model the process as its operating conditions change, allowing users to analyse the effects of such changes on the plant. Optimisation is a branch of simulation used to determine the optimum input variables for a given process. The process can be steady state or dynamic.

Frequently, models used for simulation or optimisation use the same underlying set of variables and equations, the difference being in the number of control parameters chosen and the addition (or selection) of an objective function in the optimisation case. In simulation, sufficient control parameters will be chosen such that, in combination with the equation set, the system state can be found. This is known as a 'square problem', the sum of the number of control parameters and the number of equations is equal to the number of variables. In optimisation, this sum is less than the number of variables, allowing the optimisation tool to determine the optimum state of the system within the boundaries specified by the control parameters and equation set. The system is evaluated on the basis of the objective function, frequently a total cost for the system as currently specified.

Formal descriptions of computer modelling from a mathematical viewpoint tend to avoid the concept of control parameters. The system is represented by a set of variables and associated equations and any variable can be specified (chosen as a control parameter). When modelling real world entities however it is unrealistic and often inappropriate to choose many of the system variables as control parameters. The user requires certain behaviour from the system in terms of performance. This is most easily achieved by choosing system input or output variables as the parameters, allowing the model to handle the internal complexities.

One field that makes extensive use of such models is the chemical process industry.

1.5.1 Computers in Chemical Engineering

Computers are used throughout the process industries to support the design and operation of process plants. This is largely due to the increasing costs associated with these activities and the resulting need to maximise the efficiency of the processes involved. Chemical processes are generally well understood and therefore lend themselves to an equation based modelling approach. This makes them easily modelled by computers and allows a relatively high degree of confidence in the end results. Where processes are less well-understood, such as in biochemical engineering and complex reaction kinetics, equations can still be used to represent conservation laws and other areas of the process as understanding allows.

Applications

The five main applications of such models within the industry are:

- Design of new plants (synthesis);
- Upgrade of existing plants (retrofit);
- Optimisation of operating parameters;
- Plant control;
- Operator training.

Synthesis and Retrofit

Computer Aided Design (CAD) methods are common within the process industries. As the underlying technologies become more complex and the economic and environmental operating conditions become more restrictive, optimal plant design becomes

increasingly vital. Synthesis and retrofit use a combination of simulation and optimisation tools to produce and evaluate plant designs without having to build small scale prototypes. Increasingly, the plant designs themselves are automatically generated in packages such as CHiPS (Fraga and McKinnon, 1994), allowing the designer to concentrate on the more abstract design requirements and direction. Knowledge management and decision support tools to support the design process are also under development (Banares-Alcantara et al., 1994). Frequently, these packages require access to modelling capabilities.

Optimisation

Chemical plants typically have life spans measured in decades. Factors such as raw material and utility costs and the value of the end product will change over this period. It is therefore often necessary to alter the operating parameters of the plant in order to maximise profits against a changing set of operating costs. Optimisation tools can be used to determine the best set of operating parameters.

Plant Control

Computers have replaced plant operators as the direct controllers of the chemical plant. Whereas operators used to have to watch gauges and adjust valves accordingly, computers now monitor the state of the plant and alter the operating conditions to achieve the state dictated by the operators. In advanced control systems, the operating conditions are optimised to fit the desired operating state, the current operating costs and, in some cases, even the current weather conditions. Such systems require accurate models of the plant and better than real time optimisation methods in order to be effective.

Initial design and validation of these complex control systems is often performed using dynamic simulations. This allows the control system to be tested across the full range of operating conditions and emergencies that it could be exposed to.

Operator Training

Dynamic simulations are also used to assist in operator training. This allows the examiners to test the operators reactions over a wide range of scenarios which might be too expensive or dangerous to perform on the real plant. Mistakes do not result in expensive and fatal accidents and there is no requirement to take the plant off normal, productive operation. Training is therefore safe, relatively inexpensive and easy to perform.

1.6 Integrated Process Models

Traditionally, equation based models have been constrained by the limitations of the solution and optimisation methods to look at relatively small models. Such models would typically involve either detailed modelling of sub-sections of a plant or more broad brush modelling of the plant as a whole. As the available methods have developed, there has been a move to produce more complete models of the process, sometimes modelling several plants and their connections within a single model. These integrated process models are extremely powerful, allowing much wider scope for optimisation, both at a design and an operational level.

Modern process plants are frequently highly coupled to other plants. The output from one plant may be used in whole or in part as the feed-stock to another. Therefore, optimal operating conditions for both plants individually may not in fact be the optimum for the coupled system. Such interactions are generally too complex to handle mentally and the savings possible are only identifiable by use of such models. Utility systems, the systems that provide heating, cooling and electrical power around the plant tend to work on a site wide basis. The design of these systems is realistically only possible using optimisation based design methods and detailed, integrated process models. The development of such design methods is an active area of research, driven by the large savings achievable through effective energy integration.

1.7 Project Overview

The project resulted in the development of a novel modelling environment, intended to support the development of complex models and the evaluation of novel solution, optimisation and processing methods. The requirements, existing systems and the environment are outlined below.

1.7.1 Project Requirements

The work of LSOG required an equation based modelling tool capable of:

- Developing and building complex, large models;
- Accessing and manipulating the resulting large variable and equation sets;
- Easy addition of new solvers and other modelling tools;
- Expansion to include new data structures as necessary;
- Ability to run as a stand-alone package;
- Ability to run under existing packages.

1.8 Proposed Modelling Environment

The environment provides sufficient capability for the user to describe the model in terms of a variable set and a set of methods with which to manipulate the variables. Some of these methods will describe equations but the handling of all processing is performed outwith the modelling environment by external functions. These functions act as agents, linking the modelling environment to external systems. Separating the description of the model from its processing allows the complexities to be dealt with in a full programming language (external functions are written in Fortran90 or C). The behaviour of the system is tailored by the user, the modelling environment existing

solely to store the model structure and provide the interface layer between the external systems.

The proposed environment consists of a model description language and a three tier architecture providing;

- Model building routines;
- Core data structures;
- Application and Method server.

Model building routines provide the capability to read and parse input from either user written data files or interface routines within external packages, to construct a process model from this data and to manipulate the variable and equation set. The elemental model data, defining the variable and method set, and the current sub model library are stored within the core data structures. The application and method server stores the model data for the active model and interface routines to the application and method routines. Model data is passed from the core data structure to the application and method server using standard array formats.

While the environment is currently operating on a single machine, the architecture allows the use of a thin client approach. Webopedia, the online computer encyclopedia defines a thin client as:

> In client/server applications, a client designed to be especially small so that the bulk of the data processing occurs on the server...
>
> Although the term thin client usually refers to software, it is increasingly used for computers, such as network computers and Net PCs, that are designed to serve as the clients for client/server architectures. A thin client is a network computer without a hard disk drive, whereas a fat client includes a disk drive.

The client consists of the Java GUI, model building routines and the core data structures. This set requires very little memory or processing power in order to function and can therefore operate on a relatively low powered and inexpensive computer. Storing the application and method server on a remote machine allows access to a more powerful machine when required (typically during the numerical processing of the model) and spreads the costs of expensive software licenses across the user group.

1.9 Chapter Summary

The thesis is structured as follows:

- This introduction;
- Requirements for a modelling environment and review of current technology;
- Overview of the Flexible Modelling System (FMS);
- Development of the FMS and its data structures;
- Using the FMS; modelling, extending the tool-set and language;
- A proposal for an initialisation method for object based modelling languages;
- Discussion;
- Conclusions;
- Appendices containing example models and user guide.

Chapter 2

Requirements of a Modelling Environment

2.1 Requirements Engineering

Requirements Engineering provides software developers with an abstract definition of the system. The first phase of the process, capture of the user requirements, involves detailed discussion with the intended end user(s) to determine what the system must provide. This should specify the environment (domain) in which the application is to work and the capabilities that the system should provide. This stage is followed by derivation of the system requirements. These provide a functional breakdown for the system, outlining the major processes, components and modules needed to satisfy the user requirements.

If done properly, the user and system requirements provide a timeless, solution independent statement of what the user needs (rather than wants) a system to be capable of and the major processes and components required to satisfy these needs. This allows future systems to be developed against this set of requirements, the solution arrived at reflecting the technology at the time and the users' priorities.

The following chapter applies these principles to produce a set of key user requirements for a numerical modelling environment. These were derived through discussion

with colleagues in both the LSOG and the wider modelling community, review of the literature and existing applications and the honours students who worked with the environment at different phases in its development. The approach taken has therefore been evolutionary; feedback from the user has been incorporated at each stage in the development and resulted in a new version being released.

2.2 Problem Domain

The application must be capable of supporting the development, processing and analysis of large, numerical models. Users will predominantly be research staff with some degree of computing background and a firm grounding in the principles of numerical modelling and equation-based modelling in particular. Provision should be made however to support more routine use by undergraduate or less experienced users.

As new numerical methods, databanks and analysis tools become available the system must be capable of being extended to incorporate these.

Two terms that require definition are method and application. A method is a user written function that performs some processing step based on the current equation and variable set. An application is an external piece of software such as a NLAE solver or optimisation routine that is linked to the environment.

2.3 User Requirements

The user requirements are presented in Figure 2.1.

Requirement	Description
1	**Develop and build complex, large models**
1.1	Define new models
1.2	Re-use existing models
1.3	Combine models
1.4	De-bug models
1.5	Dynamic or steady state models
2	**Access and manipulate the model data**
2.1	Locate specific variables or groups of variables within the variable set
2.2	Locate specific equations or groups of equations within the equation set
2.3	Alter variable (value, bounds etc.) and equation properties
3	**Apply methods and applications to the model**
3.1	Extract subsets of the variable set and associated equations
3.2	Select methods/tools to apply
3.3	Apply the method/tool
3.4	Interact with the method/tool as appropriate
3.5	View the results
4	**Add new methods and applications**
4.1	Access the core model data
4.2	Define an interface between the modelling environment and the method/tool
4.3	Define a control routine to access the method/tool
4.4	Call that interface when required
5	**Add new data structures**
5.1	Access and add to the core model data
5.2	Add to the model definition language to incorporate the new data structures
6	**System Use**
6.1	As a stand-alone application
6.2	As a modelling facility within a system

Figure 2.1: High Level User Requirements for Modelling Environments

2.4 Implications of the User Requirements

2.4.1 Develop and Build Large, Complex Models

Define New Models

The ability to define new models requires a model definition language (MDL). This language must be capable of describing the variable set required to represent the model and the relationships between those variables (generally the equation set) (Westerberg and Benjamin, 1985) and (Marquardt, 1994). For each variable it must be possible to define:

- Initial Value;
- Upper and Lower Bound;
- Type of variable (integer or real) (Barton, 1992).

In addition to defining the variable set, the user must be able to define the relationships between those variables. With the intention of supporting an equation based modelling approach, these relationships will typically be represented as equations of the form:

$$A(\bar{x}) = B(\bar{x}) + D \tag{2.1}$$

This relationship is translated into standard form, $F(\bar{x})$, as:

$$F(\bar{x}) = A(\bar{x}) - (B(\bar{x}) + D) = 0 \tag{2.2}$$

From this form, the system must be capable of deriving residual and first and second derivative values for the current variable set $\bar{x}$. Residual and derivative data is required in order to use Newton's Method for NLAE solution and optimisation methods.

Given this structure it is possible to represent both real and integer variable problems and define algebraic and differential algebraic equations, specifications and objective functions. Languages such as Gproms (Barton, 1992) allow the direct representation of partial differential and algebraic equations within the language and recent extensions (Oh and Pantelides, 1996) have expanded this to incorporate integrals as well. The other approach, as adopted by the Engineering Design and Research Center at CMU during the development of ASCEND III (Piela, 1989) and ASCEND IV (Allan, 1998), is to represent the PDEs using algebraic equations.

Many complex models contain sets of equations with the same form. Examples of this include component balances across a mixer model or, as illustrated below, pressure balances across a distillation column:

$$P(2) = P(1) - DP; \tag{2.3}$$

$$P(3) = P(2) - DP; \tag{2.4}$$

$$P(NTRAYS) = P(NTRAYS - 1) - DP; \tag{2.5}$$

Where NTRAYS is the number of trays within the column, P(I) the tray pressure for tray I and DP is the inter-tray pressure drop.

Specifying each of these equations and having to add or remove equations from the model to suit the number of trays within a specific column is time consuming and minimises reusability of the model. Use of a do loop structure with a variable end, and possibly start and step size, value combined with the ability to locate sets of variables through some search criteria allows the same set of equations to be declared much more efficiently as:

```
FOR I = 1 , NTRAYS - 1
 P(I+1) = P(I) - DP;
```

Complex models can incorporate discontinuous functions, as illustrated in Figure 2.2, such as the relationship between vapour fraction and steam quality for a single component stream. Where the solution lies close to such a discontinuity, successive solution steps may iterate between states on either side of the discontinuity. Use of different equations below, at and above the discontinuity can address this issue ((Zoppke-Donaldson, 1995) and (Ricoramirez et al., 1999b)). The equation used at the discontinuity provides an artificial bridge between the functions above and below it, creating an approximation to the original function that is continuous both as a function and across its derivatives.

This requires the system to be capable of using a different equation depending on the current variable values and typically takes the form of a simple IF-THEN-ELSE structure. The ability to alter the equation state based on variable values is present within gPROMS and has recently been implemented in ASCEND IV (Ricoramirez et al., 1999a) as Conditional Blocks.

This structure also supports the modelling of dynamic systems. In order to model a dynamic process it is necessary to solve the model at an initial state. This is achieved by replacing the differential equations with initial conditions at Time = 0. Use of a

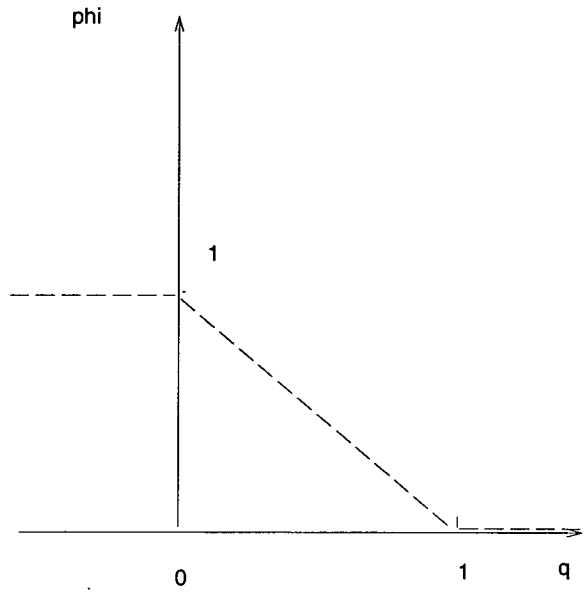

Figure 2.2: Plot of Vapour Fraction vs Quality for a Single Component Stream

conditional block within the equation declaration assists the modeller in constructing the appropriate set:

```
IF ( Time = 0 ) Then
Use intial conditions + AEs
Else
Use DAE form;
END IF
```

As models become more complex, advanced users of the environment may wish to embed analysis, debugging or output functions within the generic model. These functions do not directly manipulate the variable or equation set but allow the user to automate standard tasks (such as output) or assist with traditionally time consuming tasks such as debugging or analysis. Since the function is embedded within a generic representation of the model the function must itself be generic. This leads to effective code reuse and a truly customisable modelling language. The functionality of the language is developed by the user community on a modular, task driven basis rather than provided by a separate development team.

Reuse / Combination of models

There are two aspects to model reuse:

- Reuse of a model that the user has written using the MDL;
- Reuse of an external model.

The first issue is trivial from a modelling langauge viewpoint. Users will generally wish to reuse a particular model that they have created within the system, using the MDL. This is achieved by storing the model description in a permanent file in a format in which it can be retrieved at a later date. The ability to reuse a given model is greatly enhanced if the modelling language allows the development of generic models. As an example, a generic model of a distillation column would allow the user to specify the number of trays within the column for each column in the process, rather than requiring different models for each possible column configuration. Reuse by subsequent users however relies on adequate documentation existing describing how the model was written, what it does and the assumptions made at the time (Allan, 1998).

Reuse of external models is not trivial however. Equation based models have been used within the process industries for decades. Some of these models are huge, describing complex processes and involving proprietary physical property databases. The traditional approach to modelling environments has assumed that the user will use the environment to represent the entire model. In order to do this, the existing model must be analysed, decomposed and transferred into a new format. Due to staff turnover described earlier it is unlikely that the knowledge to analyse and decompose the model will still be present in the organisation. Assuming the expertise is available, the effort required to transfer the model can often impose manpower requirements too expensive to justify and therefore the organisation continues to use the existing modelling tools.

This limits the use of novel modelling environments within industry and therefore industry's access to novel processing and solution methods. It is difficult and time consuming for the users to link models in order to perform enterprise wide simulation

and optimisation. Frequently this would involve an iterative and manual transfer of data between stand alone models in order to achieve this scale of modelling.

A truly flexible modelling environment must support the reuse of existing models (Marquardt, 1994) and must therefore be capable of supporting a mixed model format. The global model will consist of a number of sub models, some of which are written in the environments MDL and some of which may be external and accessed through an interface layer. The environment stores the model structure and current status, passing this data or subsets of it to different modules for processing.

Debugging

The remaining major factor in building large, complex models is that of debugging. As is common in large programmes, the larger the separable blocks of code to be debugged become, the more difficult the task becomes. Programming languages initially developed modules, subroutines and functions to assist with this and code reuse. The development of object orientated languages has seen a move to even more efficient methods for reuse and debugging of code and, in common with modelling systems such as ASCEND (Piela, 1989) and gPROMS (Barton, 1992), an object orientated approach is proposed. This aids debugging (Piela et al., 1991) and reuse of smaller models which can be combined to represent much larger, complex processes.

Starting from a blank model library, typical development would construct the following models, debugging each as they were developed:

- Basic stream model;
- Stream with enthalpy / pressure relationships;
- Mixer unit;
- Splitter unit;
- Etc.

As the models develop, the user describes the additional equations and variables required to model the new process but includes the already debugged, smaller models as necessary. This hierarchical approach is common in complex model development (Westerberg and Benjamin, 1985) and is especially applicable to the chemical process models under development. The hierarchical design method (Douglas, 1988) is equally applicable to design of process models as it is to design of chemical plants (Marquardt, 1996). If there is a problem with the model, this will lie in the new variable and equation set and should therefore be easier to locate.

To support development of large and complex models the MDL must allow the user to:

- Declare scalar and vector variables;
- Specify variables (both individually and by some search criteria);
- Allow the user to create sets of equations from a single statement (Do Loops);
- Allow the system to select the correct form of the equation given the current variable set or model state (Conditional Statements);
- Develop new models from existing, generalised models of process units (hierarchical development).

2.4.2 Access and manipulate the model data

As models become increasingly larger and more complex it becomes increasingly time consuming to locate the variables of interest. This is particularly the case in complex models where many variables are providing internal detail to the model, rather than being of direct interest to the user. Examples of this include component molar enthalpies and coefficients used to calculate thermodynamic and other physical properties. These values are essential in order to produce the desired level of accuracy of the results but are not of direct interest to the user who is more interested in the flow, composition, pressure and temperature characteristics of the process.

In addition to locating and viewing the required subset of the variable set it is necessary to be able to manipulate the properties of that set. Frequently, changes to bounds data or current values will need to be applied across the flowsheet. Having to explicitly state that the upper bound for each flowrate in the model is now 500 kmol/h individually can be time consuming. The ability to apply changes to groups of variables identified by a given search criteria is essential. This makes the difference between having to specify each variable individually or by use of a single command as illustrated below.

$$F(1)upperbound = 500; \tag{2.6}$$

$$F(2)upperbound = 500; \tag{2.7}$$

$$F(3)upperbound = 500; \tag{2.8}$$

$$Etc. \tag{2.9}$$

And:

$$F(*)upperbound = 500 \tag{2.10}$$

The wild-card symbol (*) is commonly used to match strings. Use of such constructs allows the user to access and manipulate groups of variables rather than specifying each change individually.

2.4.3 Apply methods and applications to the model

The previous sections have discussed the requirements for defining models, but have not considered processing methods. Separation of the model description language from the methods, applications and equation handling routines is necessary (Westerberg and Benjamin, 1985), (Stephanopoulos et al., 1990a) and others in order to avoid the close coupling displayed by most existing environments. A language closely coupled to the end methods and applications becomes too specific, in effect tailored for describing models of a particular type, described in a particular way. This and the following section consider the requirements for applying and adding methods and applications within the environment.

The concept of passing the model data to an application for analysis or other processing

is not new. The ability to pass subsets of the model data, as defined by some specified criteria, allows use of appropriate solution methods for different sub models within the global model. Such flexibility is not possible without the ability to identify and extract subsets of the equation and variable set. Without this ability the user must rewrite existing models in order to incorporate them into the new modelling environment. This requirement was initially identified by Westerberg (Westerberg and Benjamin, 1985) and a recent thesis from EDRC at CMU (Allan, 1998) included work in this area.

User defined methods, written in a programming language such as C or Fortran allow the user to embed complex functionality within the model. Broadly proposed in a thesis from CMU, (Abbott, 1996) and within this work, (Mitchell and Morton, 1996), such methods can perform:

- Analysis routines;
- Initialisation routines;
- Calls to external applications and libraries;
- Description of equations;
- Model specific output;
- Calls to non-standard format models.

The complexities inherent in such processes is hidden from subsequent users and from the model description language itself. This allows experienced users to develop models with exactly the desired behaviour while less experienced users can incorporate the methods and resulting functionality within their own models. The methods provide easy code re-use and debugging of what are traditionally considered complex routines.

Hiding complexity from the model description language may initially seem to be an excuse for not developing a 'complete' language. It can be argued however that a language can never be truly complete and that as attempts are made to add new functionality to a language it becomes less coherent and more complex to use. This

can be referred to as language creep. Broadly, there are three approaches to developing languages:

Early modelling languages followed a minimalist approach with the user being expected to be highly proficient in coding and numerical programming. The languages and applications developed were truly flexible but offered very little support to the model development process. Packages such as GAMS (General Algebraic Modelling System) (Brooke et al., 1988) and SPEEDUP (ASPEN, 1992) allowed the user to declare variable sets and define the relationships between them in problem-orientated languages, rather than FORTRAN as was previously the case. Such packages were typically created to act as an interface between the modeller and the complex input formats required by mathematical programming packages such as MINOS (Murtaugh and Saunders, 1985).

While this approach was a great improvement over the traditionally hard-coded FORTRAN models the languages themselves did not provide any structure to support the development process. This lack of a generalised type structure meant that each model was hard-coded in the modelling language and model reuse was therefore difficult.

As programming methodologies, human-computer interface design and raw processing power have developed the modelling languages have developed in parallel. Current modelling languages can be broadly split into two categories (Marquardt, 1996): modular (block-orientated) or equation-orientated.

Modular modelling languages represent the process as a number of blocks linked by input/output connectors. These connectors transfer information such as control signals, material and energy flows between the blocks with each block representing all, or part, of a process unit using a pre-defined template. Typically, the outputs from each block are determined using a sequential modular solution method, as used by packages such as ASPEN (ASPEN, 2000).

This approach allows the rapid development of complex models but only within the functionality provided by the package. These languages require little, if any, programming ability, following the idea that advanced modelling capabilities should be available

immediately within a COTS (commercial, off the shelf) product to anyone who needs it. Development of models not included in the available libraries is time-consuming and expensive, due primarily to the need to involve a commercial development team with different priorities to the individual user.

Equation-orientated languages allow the user to define their models at an equation level. This provides maximum flexibility in terms of the model behaviour but relies on the user having sufficent knowledge of equation-based modelling. General equation-orientated modelling languages such as ASCEND, gPROMS and DIVA (Croner et al., 1990) are purely equation based, representing the model as a set of equations and variables. While ASCEND does check for dimensional consistency within equations, general modelling languages can be defined as those that contain no application specific features. These languages are object-orientated and support the hierarchical decomposition of models aiding model reuse and modification.

Process modelling languages aim to assist the developer by incorporating application specific knowledge in the model development process. MODEL.LA (Stephanopoulos et al., 1990a) and (Stephanopoulos et al., 1990b) is an example of such a language. Models are developed in a hierarchical structure from building blocks describing entities such as components, mixtures, phases and streams. Other blocks define transport mechanisms for energy and material transfers between phases. This approach is often referred to as phenomenological modelling. The current implementation of MODEL.LA, (Bieszczad et al., 1999), produces gPROMS input files from the MODEL.LA representation. The approach is similar to that behind GAMS in providing an alternative interface to an underlying modelling and solution/optimisation tool. This approach has been developed to investigate less structured modelling of complex processes where the number of phases can change (Perkins et al., 1996).

As these modelling environments develop, they become increasingly powerful and capable of representing more and more complex relationships and processes. Development of such systems to incorporate a new structure or form of relationship is usually a research project in its own right - there is a need for an environment in which to rapidly

develop and evaluate novel applications and processing methods.

In truly advanced modelling the user will require functionality not included in a standardised package. This leads to frustration while trying to implement a type of function or model in a language not designed to support it. The basis to the approach outlined in this thesis is that a user involved in modelling at such a level will be capable of programming and will have a reasonable background in the techniques involved.

The assumption that the user will be capable of developing small functions (methods) in a traditional, high level programming language while representing the core model data in a standard and minimal model description language avoids the problem of language creep. This approach is reminiscent of that used in earlier packages such as MASSBAL (SACDA, 1993) and ASCEND II (Locke and Westerberg, 1983) but when combined with the hierarchical and object-orientated techniques discussed produces an extremely adaptable modelling environment. The environment develops to suit the users' needs, rather than the developers', while providing the necessary structure to support model development and reuse.

Westerberg (Locke and Westerberg, 1983) states that the modelling environment should be usable by both the model developers and less experienced users. This opinion has recently been repeated in a review of current ecological modelling techniques (Lorek and Sonnenschein, 1999) but it is interesting to note their opinion that different tools are needed to satisfy different users. While Westerberg's statement represents the ideal, Lorek's requirement for tailored, application specific, modelling tools is probably realistic.

2.4.4 Add new methods and applications

Adding new methods and applications to a modelling environment requires access to the core model data in a format that is readily adaptable to the requirements of the specific function. The principal issue here is one of documentation; sufficient information must be available to allow the user to locate and add to the appropriate areas of code.

Many packages impose a standardised 'look and feel' across all applications. This requires the user to have a thorough knowledge of the programming language and application structure in order to add new applications. For complex applications this can be extremely time consuming, involving large changes to the model data and application window in an attempt to include most, possibly all, of the applications functionality. A better approach is to call the applications' driver routine directly and interact with it through its own interface rather than an environment standardised one. This allows the user access to all of the applications functionality, rather than the subset which has been mapped into the calling environment. Methods however should be accessible through a standardised interface from within a model definition.

2.4.5 Add new data structures

Addition of new data structures allows frequently used data types to be incorporated into the language as required. Again, the principal issue is one of documentation. In an open structure however, data can also be held in user defined external files. Typically used for control data for solvers and similar tools, such files can be used to store model specific data where the language does not currently support the functionality.

2.4.6 System Use

Traditionally, modelling environments have been developed as stand-alone packages. Frequently however, users want to incorporate modelling capability within existing applications such as CAD, flow-sheeting and design support tools such as KBDS (Banares-Alcantara et al., 1994), N-DIM (Westerberg, 1997), DESIGN-KIT (Stephanopoulos et al., 1996) and Epee (Ballinger et al., 1993). With a stand-alone modelling package, data must be manually exported between the two (or more) packages. This is inefficient and often introduces errors. As a result, the capabilities provided by the modelling tool are not fully exploited and the overall process suffers.

It should be noted that the ASCEND modelling language is embedded within a wider

modelling environment comprising solution and analysis tools. ASCEND IV has recently been incorporated within the N-DIM process design environment. gPROMS, as discussed earlier, provides the modelling capability for MODEL.LA and is moving towards providing an embedded modelling capability. At the time of writing however, there are no truly open modelling environments supporting the use of mixed format models and allowing data transfer between peer applications.

Modelling environments should be capable of both stand-alone and embedded use within another application. In order to run effectively within another application, the functions required to build, process and retrieve models must be accessible from the host application. These functions must be capable of running without user intervention being required to select solution methods, start processes, etc. In this scenario, the model will typically be defined and initialised within the host application. the environments sole responsibility is to return a converged solution for the model as it is currently specified.

2.5 Overview of Existing Systems

2.5.1 Modelling Environments

Traditionally, modelling applications, and associated methods, have been developed by individuals, be they academic institutes or corporations, without a perceived need to link to external systems. Sometimes this is a result of commercial considerations - the application may have commercially sensitive information regarding the processes involved in the plant within it but more often is a result of the technology available at the time.

Common modelling environments include:

- Bespoke programmes;
- Spreadsheets;

- Flow-sheeting Packages (ASPEN, HYSYS(Hyprotech, 1995));
- Equation based modelling languages (GAMS, ASCEND, gPROMS)

Many current process models are written as bespoke packages in languages such as Fortran. These have a number of advantages; the application is tailored precisely to the customer's needs and can easily access proprietary databases. Such models are however notoriously difficult to maintain due to staff turnover within the operating life of a plant or to update to reflect changes in the plant structure or conditions. The application is typically written as a stand-alone, complete modelling package. Access to external or new modelling tools is therefore extremely difficult, if not impossible.

For simple models, spreadsheets often suffice. Packages such as MS Excel are increasingly powerful and readily available, providing access to plotting, basic solution and optimisation methods and a degree of customisation through the use of macros. Truly complex models are however difficult to create or alter and such packages are still inherently stand-alone, minimising their access to external methods or their use by wider systems.

Flow-sheeting packages such as ASPEN provide the user with large model and physical property libraries, accessed through powerful and flexible user interfaces. Process models are constructed from standard building blocks provided within the package.

The prime limitation of such systems is the inability to add models as required - if the model or method does not exist within the package then there is no way for the user to add it. Development of such packages is therefore incremental, driven by the commercial priorities of the developers. This limits their applicability in any environment where new models and methods are constantly being developed.

The most flexible environments are provided by equation based modelling languages such as ASCEND and gPROMS. These allow the user to build new models as required, to reuse or adapt existing models and to connect new applications. They are however stand-alone packages which aim to provide a 'one stop shop' for a users' modelling

requirements and are not easily extended to include new data structures due to the close coupling between the modelling language, model storage and application set.

2.5.2 Modelling Tools

As described above, most existing modelling environments come with a fixed application library. This allows a coherent user interface between the applications but often means that it is difficult, if not impossible, to add new applications into the library without a thorough knowledge of the environment, the language it is written in and the interactions with other systems.

Particularly in a research environment, the tool set required changes rapidly. As new tools and methods become available, the type, size and complexity of viable problems increases. In order to perform timely work there is a requirement to incorporate and use these new methods as they are developed; integration must be as quick and simple as possible. This type of user is generally expert in their field and highly computer literate and therefore does not require the level of support that a common user of such packages needs. Accepting that the user will need to do some, albeit a minimum amount where possible, of coding allows the system to be much more powerful and easily extended at the expense of some 'usability'.

Novel tools can require data structures and associated handling routines not found in existing environments. The environment must be adaptable to incorporate these changes where necessary. Given the continuous development of new methods and tools and the resulting changes in the modelling environment's data structures it can be argued that the existing trend to provide complete modelling environments (modelling language, model handling routines and application library) as a package, albeit with some ability to access external systems, is too restrictive.

2.6 Summary

The key to a truly flexible modelling environment is the users' ability to tailor the system and add functionality as required. While this does require the user to be a reasonably competent programmer, familiar with the processes involved, the increase in flexibility far outweighs the additional skills required. A modelling environment, or any other application aimed at such a complex and developing area, that intends to provide a COTS solution for all user applications will fail.

As no existing modelling environment completely satisfied the requirements outlined it was decided to develop an in-house solution. The following three chapters describe the functionality, structures and use of the environment developed, the Flexible Modelling System (FMS).

Chapter 3

JFMS Functionality

3.1 Overview

JFMS provides an extensible modelling environment based around a core model definition. It can operate both as a stand-alone package and as a modelling capability within a host application. Models are developed using an object orientated modelling language and can be linked to non JFMS format models. Functionality is provided by user written methods rather than being declared within fixed language constructs as is typical with other modelling approaches. This allows the environment to be tailored to suit the needs of the users, rather than the developers.

3.2 Core Model Definition

JFMS is intended to act as an interface between mixed format models, databanks and applications. In order to achieve this a core model definition has been developed. This represents the model structure, and current state and provides links to the methods required to analyse and process the model.

The core model definition consists of:

- A list of model components (units, streams and sections within a process model for example);
- A variable vector (value,upper and lower bounds);
- A specification set;
- A method set.

This format is common to both the GUI and the application server. The behaviour of the model is controlled by the methods and it is possible to extract subsets of the variable and method sets from the global model for separate processing. The global model in effect controls the links between the different sub-models.

3.3 Model Description Language

The Model Description Language developed is an object orientated language declaring generic representations (ETYPEs) of component models that the user requires. Each ETYPE contains sufficient information to create the variable and method set required to represent an instance of the generic model within a specific process. ETYPEs can:

- Inherit information from existing ETYPEs;
- Declare variables (variable name , type and size, if vector);
- Declare methods;
- Declare instances of other ETYPEs;
- Link model components;
- Alter variable values;
- Declare initialisation methods (see Chapter 6).

3.3.1 Methods

Methods are used to provide the model functionality and are declared within an ETYPE using a standard interface. Methods can be used to:

- Provide model analysis;
- Initialise models;
- Call external applications and libraries;
- Provide an interface to non JFMS format models;
- Declare equations;
- Perform model specific output.

Each method is a user written subroutine in a high level programming language such as C or Fortran. The method takes a subset of the global model variable set and performs some processing on it. By writing methods in a standard programming language the user is not restricted to a set of functions or features that the developers have provided.

3.3.2 Application Server

While methods provide component level functionality, the application server acts as an interface between the model and the more general solution, optimisation and other processing applications. Each application is linked to the core model definition through an interface function that converts the core model definition to the format required by the application. Control then passes to the application and it's own interface. Once the processing is complete, the data is converted back to the core model definition format and control returns to JFMS.

3.4 Comparison with Existing Systems

This section provides a comparison between JFMS and existing packages. This is represented in tabular form in Figure 3.1. The major contributions are:

- The ability to provide an embedded modelling capability within other tools, without user input during the creation or solution steps (as would be required with gPROMS);
- The ability to act as a broker, connecting existing non-JFMS models to JFMS (reuse of existing models);
- The ability to embed analysis, output and any other desired functionality that can be expressed in Fortran90 within a generic model (methods);
- The ability to call external functions from within the model;
- The ability to select and process subsets of the variable and equation set, decomposing the model by unit or other criteria (see Appendix D). This allows pseudo sequential modular (block-wise) solution and multipurpose models to be defined.

  As an example, shortcut and rigorous models could be provided within a generic model. Selection of the required version is achieved by selecting which methods are active. This would support the use of shortcut methods to initialise or rapidly evaluate alternative models with the ability to switch, within the flowsheet, to the rigorous version when desired.

While JFMS may appear to be a variant of gPROMS and ASCEND, a similar level of functionality is provided through a much simpler modelling language. This language represents the model structure, public variable set and the methods required to generate equations, provide analysis functions etc. General model development however is supported by a structured, object-orientated language such as in gPROMS and ASCEND and connections between units are handled in a similar manner.

Capability	JFMS	ASCEND	gPROMS	ASPEN
User defined models	Y	Y	Y	N
Attach new solvers and applications	Y	Y	Y	N
Equation-based, simultaneous solution	Y	Y	Y	N
Sequential Modular solution	Y	N	N	Y
Stand-alone modelling capability	Y	Y	Y	Y
Provide embedded modelling capability to other tools	Y	N	U/D	N
Connect existing non-JFMS models	Y	N	N	Y
Symbolic differentiation of input files	N	Y	Y	N/A
Embed analysis and other functionality in generic models	Y	N	N	N
Large number of available tools	N	N	Y	Y
No coding required	N	Y	Y	Y
Dynamic modelling support	N	N	Y (Tasks)	N
Use of section variables and wild-cards	Y	N	N	N
Call external functions	Y	N	Y	N

Figure 3.1: JFMS functionality against that of existing systems

The gPROMS and ASCEND languages have also tried to represent the detailed mathematics in the model within the language. This has lead to the language being continually developed by the original research group in order to support additional functionality.

ASPEN in contrast provides a fixed model library which the user interacts with through a graphical user interface. This allows the user to select models, connect them together and provide values for the model parameters. The user has no control over the internals of the model; it is in effect a black box.

A system such as JFMS derives all of its functionality from user written routines interacting through a standard interface. While this requires programming on the part of the user, the model is constrained by the limitations of a complete, high-level programming language such as Fortran90, rather than those imposed by the developers.

The approach is therefore a hybrid between the highly tailored but inflexible FORTRAN based approach and the structured modelling capability provided by gPROMS and ASCEND. This provides the benefits of both approaches at the expense of some user programming.

Chapter 4

Flexible Modelling System

4.1 Introduction

There are many modelling languages in existence (see Chapter 2) and therefore the immediate question to answer is "Why develop another one?" The aim of the project was to produce large, equation based models, primarily based on industrial scale chemical processes. These models would be used to test novel solution and optimisation methods developed in the first instance by researchers at the University of Dundee and allow comparison with existing benchmark methods. The principal requirements for the modelling environment used were therefore ease of adding new applications and the ability to alter the data structures used. This latter feature allows application specific data to be stored at the model building block level rather than the model level. This is necessary to improve the efficiency of the modelling process, a particular concern when building large models.

There is therefore a need to have access to the source code for the package to alter the code to implement changes in the data structures used and the ability to add applications, not necessarily solution or optimisation methods, to the modelling environment. This requires a good knowledge of the code involved and the code being written in such a way that the desired changes are possible. Two possible directions were identified; use of an existing package or development of an in-house one.

Existing packages come in two flavours, commercial and academic. Most commercial packages are monolithic in nature, improvements are made and new applications connected through new versions of the package. Such changes are generally infrequent, are always profit driven and the software involved is usually proprietary. A commercial package therefore does not allow the rapid development and testing environment required. Many of the academic packages are under continuous development but the direction of that development is dictated by the research interests of the people involved. Again, there is an inability to alter the code to explore areas of individual interest, either through lack of familiarity or the sheer scale of changes required to implement the changes desired. Since there is no guarantee that the surrounding code is in a stable form there is no guarantee that changes made for one version will work in later versions of the package.

As a result of these issues, it was decided to produce an in-house package. The Flexible Modelling System (FMS) developed as part of this project provides a powerful, user extendable application allowing rapid creation and manipulation of large, equation based models. This has been achieved by a combination of the modelling language, the graphical user interface and the data structures used within the application. The work has been performed in conjunction with Dr. Sven Leyffer of Dundee University Mathematics Department.

The development and features of this application are discussed in the rest of the chapter.

4.2 Development Path

Originally intended to provide a quick way of producing equation based models of utility systems in order to test novel solution methods, FMS has developed into a full modelling language in its own right. Whereas early versions had hard coded routines storing modelling data for a set number of unit types, later versions added the ability to describe new unit types, allow dynamic modelling, link external packages and allow user interaction with the model. This latter feature was initially implemented as a command

line driven module; the latest version uses Java and the Java Abstract Windowing Toolkit (JAWT) to allow analysis and manipulation of the model through a graphical user interface.

4.2.1 Utility System Modelling Package

The earliest incarnation of FMS was the Utility System Modelling Package (USMP). This provided a set of routines to produce Jacobian, Hessian and residual data for a set of equations and variables. The package read the problem structure from a text file, built and attempted to solve the resulting model and then wrote the results to file.

The set of equations was produced by a series of hard coded routines, one for each unit type, called in sequence from the plant unit list (PUL). Connections between units were represented as equalities between the relevant inlet and outlet stream variables.

USMP Structures

The structures used in USMP form the basis for the more flexible structures used in later versions of FMS. There are five areas of interest, these being:

- Unit Type Routines
- Plant Unit List
- Variables
- Connections, Specifications and Assignments
- Derivative and Residual Storage

Unit Type Routines

Unit Type Routines (UTR) provide the information required to calculate Jacobian, Hessian and residual data for a given instance of a particular type of unit. In the case

of USMP, UTRs existed for a range of unit types typically found in utility systems. These included:

- Stream (vapour, liquid and mixed phase, single component)
- Mixer / Splitter
- Pump / Turbine
- Valve
- Heat Exchanger (Condenser, Boiler variants)

Figure 4.1 shows an example of a mixer unit with 2 input streams and 1 outlet. In order to allow models for a given unit type to work for all types of stream (water (liquid, vapour and mixed) and air) the model is written in such a way as to avoid needing to know which type of stream it is dealing with. In order to model the process completely, 3 stream instances must be created (S1, S2 and S3 in Figure 4.1). These instances provide the additional variables and equations required to calculate enthalpies etc. In.1, In.2 and Out are therefore simplified stream models created by the mixer model and contained within it. Given that the streams modelled in USMP are strictly

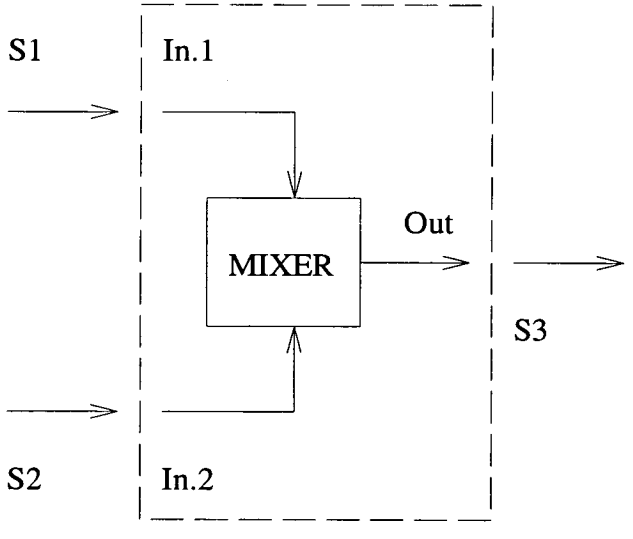

Figure 4.1: A Simple Mixer Unit

single component and the assumption that the mixer streams are at equal pressure,

- Flowrates for streams In.1, In.2 and Out (F)
- Enthalpies for streams In.1, In.2 and Out (H)
- Pressures for streams In.1, In.2 and Out (P)

Figure 4.2: Variables in USMP Mixer Model

the variable list for such a model resolves as shown in Figure 4.2. Note that specific variables are identified by combining the unit and variable names, separated by a period. In.1.F is therefore the flowrate variable for stream In.1. For the purposes of the following discussion, mixers have 9 variables, full stream instances have 4 (F,T,P and H). The mixer equations are therefore:

$$Out.F - In.1.F - In.2.F = 0 \quad (4.1)$$

$$Out.H - In.1.H - In.2.H = 0 \quad (4.2)$$

$$Out.P - In.1.P = 0 \quad (4.3)$$

$$In.2.P - In.1.P = 0 \quad (4.4)$$

Each UTR is stored in a Fortran90 subroutine. Pseudo code for the mixer model described above is shown in Figure 4.3. The subroutine looks at the relevant subset of the variable set and adds the resulting Jacobian, Hessian and Residual data to the global vectors. Arguments to the routine are as follows:

- VStart. Pointer into variable vector (x). Points to position in x one before the first variable belonging to this unit.
- x. Vector of derived type (a Fortran90 data structure) representing the variable set. Contains bounds and current value.
- Res. Real vector containing residual values for equation set.
- Jac. Vector of derived type representing the Jacobian. Contains value,row and column.
- Hess. Vector of derived type representing the Hessian. Contains value,row and column and equation.

- row. The last equation number handled.
- nJ. Last Jacobian element added.
- nH. Last Hessian element added.

Plant Unit List

The Plant Unit List (PUL) is a catalogue of the units and streams required to represent the flowsheet. For each entity (stream or unit), the PUL stores a name, unit type and pointer (VStart) into the variable vector (x). The PUL is created from the USMP input file and is used to set up the variable vector and equation data. An example for the simple flowsheet in Figure 4.1 is shown in Figure 4.4.

Variables

Variables in USMP are stored in a vector of derived type. Each element in the vector stores a name, current value and upper and lower bounds for a specific variable. After the PUL is compiled the program runs down it, totalling up the number of variables required to model each entity in the flowsheet. The number of variables for an instance of a particular type is hard-coded as part of the model library. The variable vector is then allocated to this size and its values initialised.

Connections, Specifications and Assignments

As previously mentioned, connections in USMP were handled by adding the relevant equalities to the equation set. For the model in Figure 4.1, equalities would be added to connect, for example, S1.F to MIXER.In1.F and so on (see section 4.2.1) for the complete input file. This results in duplicate variables, 2 variables exist for what is in reality only one, and extra equations to tie them together. In USMP, the user had to calculate the position of each variable in the global variable set and the equality was specified by giving the 2 variable positions in the input file. In the example given (connecting S1.F to MIXER.In1.F) since we know that the flowrates are the first variables in both the stream and mixer models we can use the relevant VStart values from the

```
subroutine Mixer

! Declare variables

! Add mass balance equation

! Pointers into x vector for specific
! variables (makes code simpler to read)

FOutP = VStart + 1
FIn1P = VStart + 2
FIn2P = VStart + 3

! Get current values for variables

FOut = x(FOutP)%val
FIn1 = x(FIn1P)%val
FIn2 = x(FIn2P)%val

row = row + 1

! Jacobian elements for mass balance
! (dFout, dFIn1, dFIn2)

nJ = nJ + 1

Jac(nj)%val = 1.0
Jac(nj)%row = row
Jac(nj)%col = FOutP

nJ = nJ + 1

Jac(nj)%val = -1.0
Jac(nj)%row = row
Jac(nj)%col = FIn1P

nJ = nJ + 1

Jac(nj)%val = -1.0
Jac(nj)%row = row
Jac(nj)%col = FIn2P

Res(row) = FOut - FIn1 - FIn2

!Repeat for other equations
end subroutine
```

Figure 4.3: Source Code for USMP Mixer Model

NAME	TYPE	VStart
S1	Stream	0
S2	Stream	4
S3	Stream	8
MIXER	Mixer	12

Figure 4.4: PUL for USMP Mixer Flowsheet

PUL (see Figure 4.4) to calculate that the variables to be equated are numbers 1 (0+1) and 13 (12+1) respectively.

Specifications and assignments are represented by a manually calculated variable id number and a real number. In the case of assignments however there is the additional ability to specify the slot (value, lower bound or upper bound) that the number should be placed in. Specifications are stored in a vector of derived type and added to the Jacobian and residual vectors when required. Assignments are not stored within the package and are solely used to overwrite the default values applied on creation of the variable vector.

Derivative and Residual Storage

In combination with the storage of the variable set, it is the storage of the resulting derivative and residual data that determines how easy it is to connect external solvers and optimisation routines to the application. In the case of USMP, all the application was designed to do was to produce Jacobian, Hessian and residual data for a given variable and equation set and transfer these to other packages. In order to do this a standard representation was required that would be memory efficient and rapidly and simply converted into the exact form required. Jacobian and Hessian are stored in standard sparse format in vectors of derived type. One element in a vector contains all the information (row, column, equation and value) required to represent an element in the required array. See Figure 4.3 for examples of using the Jacobian and Residual vectors.

USMP Input File

The following is a sample USMP Input File, setting up the model in Figure 4.1. In order to save space only one specification and two assignments have been given in order to demonstrate the relevant syntax. Only the Flowrate variables have been equated - in the complete model the pressures and enthalpies would be as well. The end of a given section of the file is indicated by the semi-colons, this was removed in later versions. Vectors were not supported.

```
ENTITY              (Note: entity name, unit type)
S1          Stream
S2          Stream
S3          Stream
MIXER       Mixer
;           ;
CONNECT             (Note: var number, var number)
1           13
5           14
9           15
;           ;
SPEC                (Note: var number, real value)
1           100.0
;           ;
ASSIGN              (Note: var number, real value, slot)
2           50.0    val
2           50.0    lbd
;           ;       ;
```

Figure 4.5: Input File for USMP Mixer Flowsheet

Summary

USMP was a useful tool and achieved what it was intended to do. However, as the project developed, the limited model library and overly user intensive modelling language became seriously limiting. The work summarised above forms the basis of the FMS packages described in the rest of the chapter.

4.2.2 From USMP to JFMS

After using USMP to construct larger models, a number of key improvements that were required were identified. These were:

- A more flexible model library.
- A more powerful language to remove all the manual calculation of connections etc.
- Addition of a user interface to allow manipulation of the variable set.
- The ability to store a model and reuse it with different values or solution methods.

As the user requirements developed, so did the language and interface that were needed to support their activities. FMS went through a number of key stages as outlined below.

- Command line driven, steady state modelling language
- Command line driven, steady state and dynamic modelling language
- Java based GUI driven, steady state modelling language (JFMS)
- Java based GUI driven, steady state modelling language with integer variable support.

Dynamic modelling was removed in the change to the GUI driven version as it was not required to support any of the work being done at the time or being planned in the foreseeable future. For the purposes of this report, the data structures and other details described are those used in the current version of JFMS. These have developed incrementally as the package has changed.

4.3 JFMS

JFMS is an extensible, generic, equation based modelling package that has been developed in conjunction with Dr. S.Leyffer of Dundee University. JFMS allows the user to add new entity types, basically generic representations of the objects the user wishes to model, be they mixers and streams or other, more application specific, objects. The current version of JFMS allows the user to quickly develop large and complex steady state models, to solve or optimise these using a variety of routines and to examine the results quickly, based on user defined search criteria. Subsets of the variable and equation sets can be extracted and passed to connected applications, allowing the package to act as a sequential modular simulator or examine blocks of the global problem. JFMS can be run either as a stand alone modelling package or as a set of routines within another application. When run as a stand alone package the user is supported by a simple to use graphical interface allowing easy manipulation and examination of the model data.

This structure allows use of the application at a number of different levels:

- By selecting already specified entities the user can build models in a manner similar to using a package such as ASPEN.
- At the next level of complexity, the user can build new entity types from a list of typical equations such as mass and enthalpy balances and basic VLE and general physical property relationships.
- The final level involves the user in adding new equations to the library to allow more complex or application specific entities to be built. At the moment, this requires some basic Fortran programming skills but this could be improved upon with the addition of an equation editor.

4.3.1 JFMS Data Flow

Figure 4.6 outlines the flow of data within JFMS. The data structures mentioned are described in more detail in section 4.3.2.

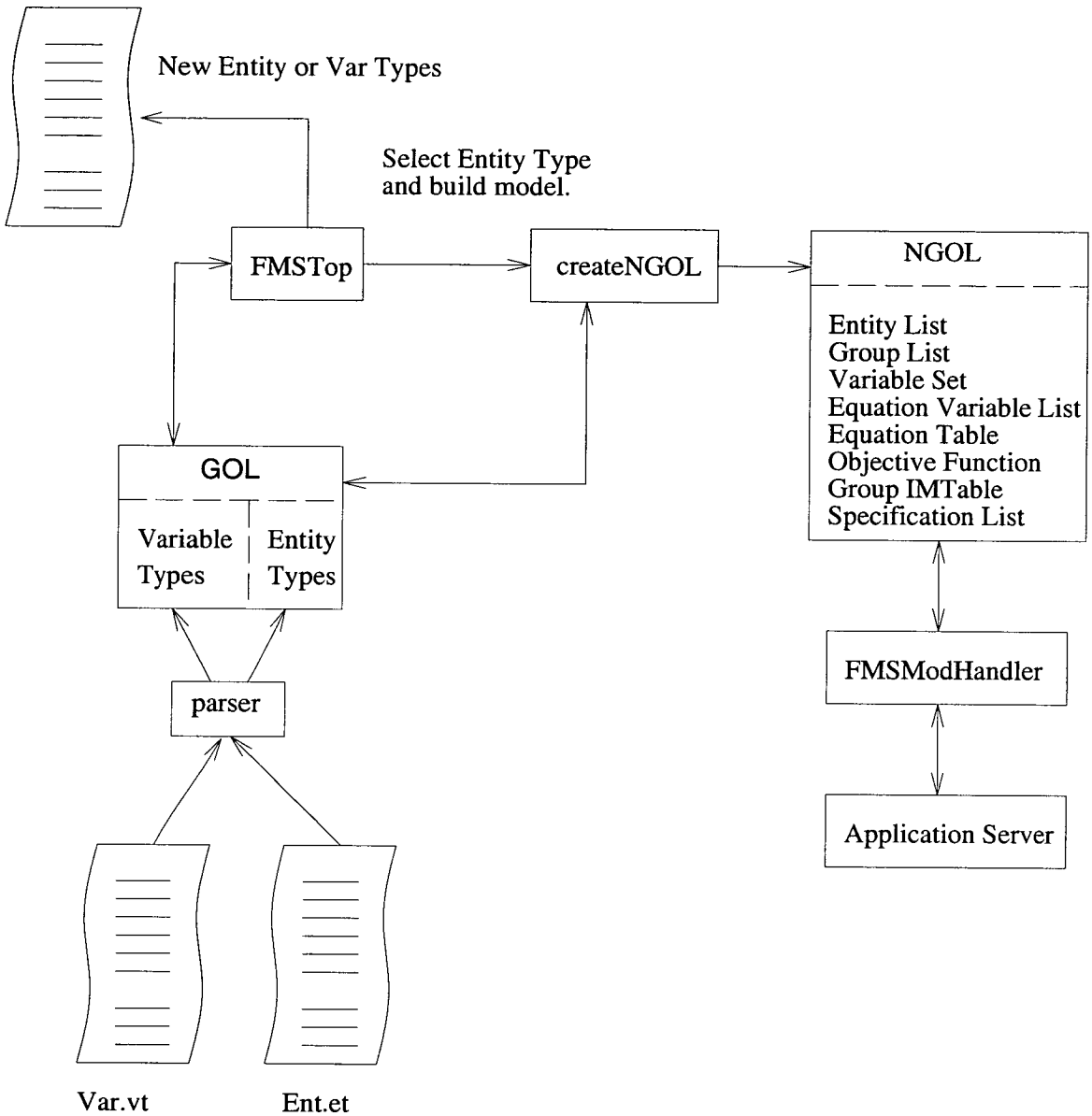

Figure 4.6: Data Flow in JFMS

4.3.2 JFMS Data Structures

There are two distinct components of JFMS: the model builder and model interface written in Java and the application server written primarily in Fortran90. In the following discussion data structures belonging to the Java side are prefixed by 'J:', those that belong to the application server by 'A:', and those that are common to both by 'JA:'. A simple mixer model is used to illustrate the steps taken in converting the model from a set of text files to the JFMS data structures. The model is given in Figure 4.11, and is illustrated in Figure 4.1. In order to reduce the size of the model, streams contain only flow and composition data and not the additional thermodynamic data that would usually be included. The initialisation method declarations are also omitted. Development of FMS models is discussed in chapter 5.

J: GOL - Generic Object Library

The GOL is the Generic Object Library. Generic objects are used to store the building blocks for models as well as the structure data for the actual models themselves. There are two identifiable classes of generic object; the variable type (VTYPE) and entity type (ETYPE). These are described below. Generic objects are stored in user written text files. JFMS reads these files, parses the data contained and stores them in the GOL as shown in Figure 4.7.

The GOL is dynamic, it can be added to or changes made to existing members during run time. VTYPEs and ETYPEs must have been declared, either within the GOL or in the current data file before they are used in other ETYPEs. Existing models continue to use the VTYPEs and ETYPEs as they were at the time the model was created.

J: GOL: VTYPE

Variable types are used to set default bounds and initial values for variables declared as being of that type. These values can be changed at a later date, either within an entity type declaration or through the GUI. A variable type is declared in a data file,

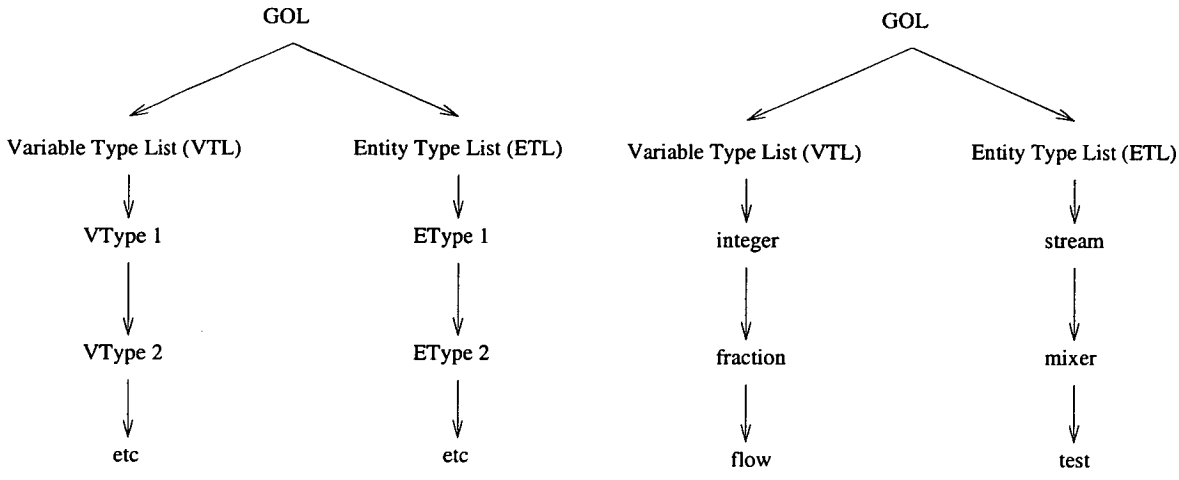

Figure 4.7: Structure of GOL and example for mixer problem

```
VTYPE     type name
UBOUND    upper bound
LBOUND    lower bound
SCALE     scale factor
VALUE     default value
END
```

Figure 4.8: VTYPE Template

```
VTYPE     flow
UBOUND    100.0
LBOUND    0.0
SCALE     20.0
VALUE     15.0
END
```

Figure 4.9: Example VTYPE: flow

this declaration taking the form shown in Figure 4.8. An example of a VTYPE for flow is shown in Figure 4.9 and is used in the mixer example in Figure 4.11.

J: GOL: ETYPE

An entity type is a generic representation of an object that the user wishes to model. To date, types have been produced for a range of objects, including streams, mixers and distillation columns although there is no limitation in the system constraining the

models to be related to chemical engineering. The entity type therefore contains all the information that the system requires to represent a specific instance of the type - typically this will involve a list of variables and related equations required, along with other data as described below.

Entity types have the following properties:

- Can inherit information from existing entity types.
- Declare variables (variable name , type and size, if vector).
- Declare equations and the variables required in those equations (see Figure 5.8).
- Declare other instances to be set up at the same time as the current entity (ie input/output streams from a vessel could be declared here).
- Create links between instances (for example to connect units).
- Give initial values or specifications for variables if different from default.
- Declare initialisation methods (see Chapter 6).

A model is itself an entity type, made up of a network of entities. Each of these entities in turn can themselves contain other entities. In Figure 4.10 a small model called MiniPlant is created which contains instances of the Mixer, HEX and Stream ETYPEs. Mixer and Hex both contain instances of the Stream ETYPEs. This nesting allows complex models to be quickly assembled. See Figure 5.20 for a generalised ETYPE data file, use of this structure for modelling is covered in Chapter 5.

It is important to make the distinction between building block (BB) ETYPEs and model (M) ETYPEs. BB ETYPEs are generalised representations of a process unit or stream. They will not contain general specifications, assignments or size data such as actual values for the number of components or trays in the process. M class ETYPEs are used to represent an actual process. Therefore, they are responsible for providing all the size data, connections between units and any general specifications the user

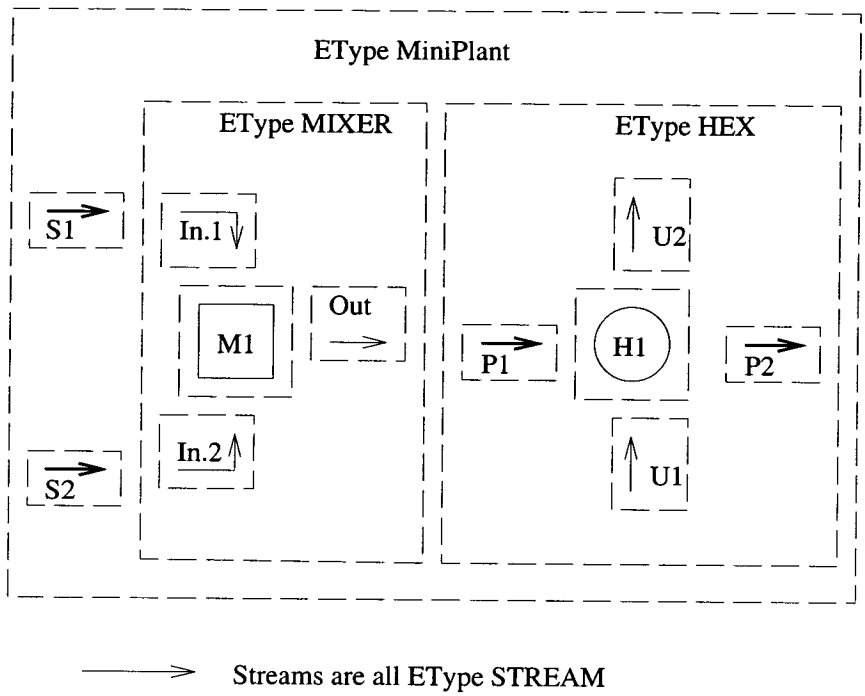

Figure 4.10: Entity Nesting in a Simple Flowsheet

wishes to include as standard for the process. Figure 4.11 shows a sample mixer model data file. ETYPEs stream and mixer are BB class ETYPEs, test is an M class ETYPE.

M class ETYPEs must be translated in order to become a usable model. In effect, an M class ETYPE provides a list of all the entities (units and streams) within the model and the necessary specifications and connections to satisfy the structural needs of the model. Typically, this latter point is satisfied by specifying values for variables such as the number of components, number of trays for a given column etc. These values define the structure of the specific model desired, replacing the generalised structure found in the BB class ETYPEs.

As an example, stream ETYPEs typically declare a variable x, of type fraction and size ncomps. This declares a vector x of length ncomps, each value of which is of the declared VTYPE fraction. This generalised form is translated when converting the M class ETYPE to the model, ncomps being replaced by the the value of ncomps for this specific model. The model produced is stored as an object of class NGOL.

```
ETYPE     stream
VARS
          F        flow
          x        fraction   ^ncomps
EQNS
          sumX
                   ^          ncomps
                   .          x.*
          ;
END

ETYPE     mixer
INSTANCE
          stream
                   in         2
                   out
          ;
EQNS
          massB
                   in.*       F
                   out        F
          ;
          compB
                   ^          ncomps
                   in.*       F
                   in.*       x.*
                   out        F
                   out        x.*
          ;
END
ETYPE     test
VARS
          ncomps   integer
INSTANCE
          stream
                   feed       2
          ;
          mixer
                   MX
          ;
CONNECT
          feed.1   MX.in.1
          feed.2   MX.in.2
FIXED
          .        ncomps     value     2.0
          feed.1   F          value     10.0
          feed.2   F          value     20.0
          feed.1   x.1        value     0.3
          feed.2   x.2        value     0.4
END
```

Figure 4.11: Simple Mixer Model Data File

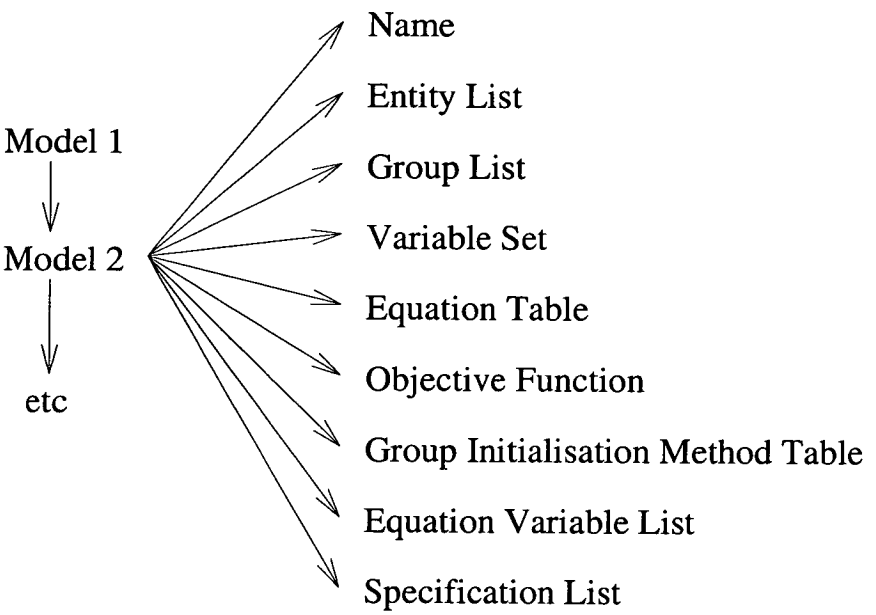

Figure 4.12: Basic Structure of the NGOL Class

J: NGOL - Non-Generic Objects

Unlike early versions of FMS, JFMS allows multiple models to be built and accessed from the interface. These models are stored in a vector of class NGOL (Non-Generic Object Library), one model per element in the vector. An NGOL is therefore a complete model, as outlined in Figure 4.12.

J: NGOL: Name

Each model is given a name to aid the user in identifying particular models from the set of available ones.

J: NGOL: Entity List

The Entity List is the JFMS version of the USMP Plant Unit List (see Figure 4.4) and is built in the same way. Due to some changes in the way connections are represented the information contained is slightly different however. For each entity created the following information is stored:

- Name.
- Group. Group number that this entity belongs to (Figure 4.13).
- Parent. ID number of entity that created this one.
- Type. ID number of ETYPE of this entity.

JA: NGOL: Group List

When a connection is made between entities in a model, USMP required a set of equality statements to link what was a duplicate set of variables. To avoid this effect, JFMS uses the Group List to store the unique entities found in the model.

In USMP, a set of variables and related equations existed for every entity in the entity list. As a unit typically specifies both its inlet and outlet streams, unique variables and equations exist within the global equation and variable set for these streams. As a stand-alone unit this creates no duplication of data. When such a unit exists as part of a flowsheet however its associated streams are connected to streams belonging to other units. There is only one stream in the flowsheet but the entity list sees two: the outlet from the first unit and the inlet to the second. JFMS maintains the flexibility of being able to access the relevant data using either unit name through the entity list group number slot. The relevant member in the group list stores the required data for the model and therefore only the necessary information is stored. This removes the need for a set of equality statements as used in USMP by forcing both units to use the same data for their connecting stream.

Reference to Figure 4.10 shows that M1.Out (the mixer outlet stream) and H1.P1 (the inlet stream to the heat exchanger on the process side) are in fact the same stream. Both M1.Out and H1.P1 will appear in the Entity List however only one entry will exist in the Group List for the pair. This means that only one set of variables and equations exist for the stream in question but that, through the Entity List, this information can be accessed by both the mixer and heat exchanger, as illustrated in Figure 4.13. While this example is obviously simplified, the savings in terms of number of equations and

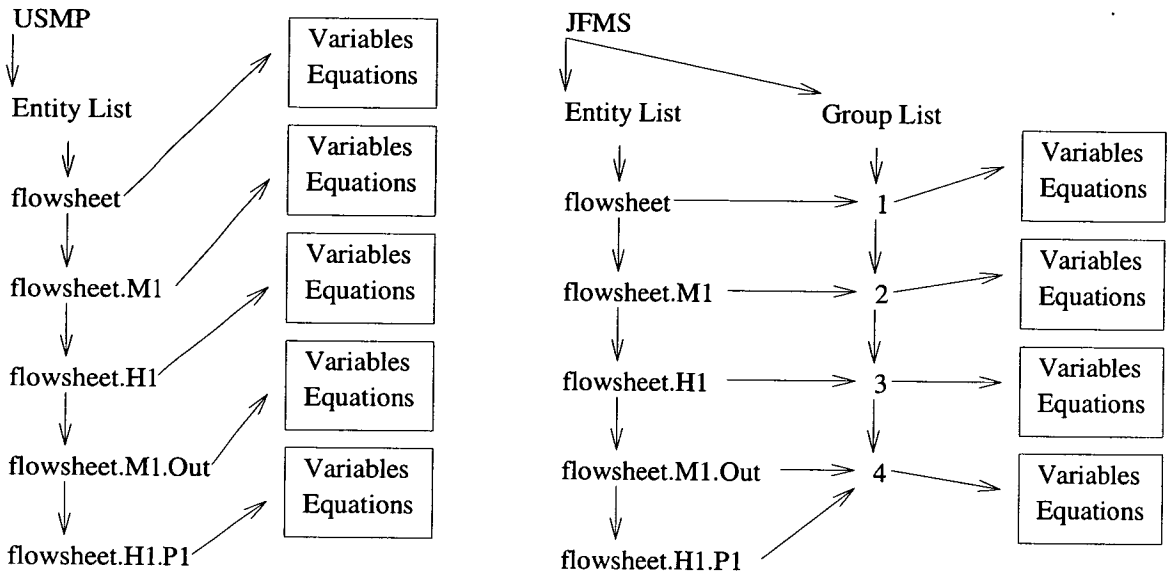

Figure 4.13: Derivation of Group List from Entity List

variables is of the order of 50% over a large flowsheet. Five entities in the entity list are in fact four unique entities in the model, as reflected in the group list. Each element in the Group List stores the following information:

- Name
- Variable start and finish indices (links to Variable Set)
- Equation start and finish indices (links to Equation Table)
- Initialisation Method index (links to Group Initialisation Method Table)

JA: NGOL: Variable Set

All variables in the model are stored in the Variable Set. Each variable stores:

- Name
- ID number of entity the variable belongs to.
- ID number of VTYPE.

- Status of the variable (specified, user assigned, default etc).
- Whether variable is active or inactive (allows extraction of subsets of the model).
- Whether the variable is continuous or integer.
- Lower bound
- Upper bound
- Current value
- Scale factor
- Initial value

JA: NGOL: Equation Variable List

The Equation Variable List (EVL) stores the locations of each variable required by the Method calling structures and is an integer vector. For a given function call a particular subset of the Variable Set will be required. On parsing the input file, JFMS records the location of each of these variables in this list. Each of the EFCs contain pointers back into this list allowing rapid access to the appropriate variable subset.

J: NGOL: METHOD CALLING STRUCTURES

JA: NGOL: Equation Table
JA: NGOL: Objective Function
JA: NGOL: Group Initialisation Method Table

- Equation Table is responsible for setting up all linear and non-linear equations but can also be used to perform other analysis or link to other models and packages;
- Objective Function stores the data required to set up the objective function if needed;
- Group Initialisation Method Table allows access to external initialisation routines.

These structures provide links to external functions. Each entry stores the data required to access the relevant function and pass it the appropriate variable set. In addition to this there is an active/inactive flag. In combination with the equivalent flag in the Variable Set this allows extraction of subsets of the model. This can be used to run the application in a sequential modular manner on a unit by unit basis or to extract sections of the model on a more mathematical based block wise basis.

Data held is therefore:

- Name
- Pointer to variable start and finish in Equation Variable List (see above)
- Whether function is active in current process.

JA: NGOL: Specification List

The final set of data required to represent a given model is the Specification List. As with USMP (see section 4.2.1), specifications in JFMS are stored using the following structure:

- Variable ID number
- Real value

Unlike USMP however, JFMS uses an entity and variable name based method in the input file. This allows a more flexible and natural method of identifying a specification, allowing the ETYPE in question to be used in many different models. The earlier method was effectively the equivalent of hard coding the entity's position in the model.

A: Derivative and Residual Storage

The storage form used in USMP was found to be sufficiently flexible to be used in JFMS. Data is stored in a sparse format and routines exist to change between different, commonly used formats.

Derivation of NGOL for Mixer Model

This section expands on the createNGOL method illustrated in Figure 4.6. This is a Java method that takes a M class ETYPE and converts this into a usable set of variables, equations and associated structures. The example used is that of the mixer process described in Figure 4.11.

The first step is to compile the Entity List. Name, parent and type are set at this stage, group ID number is assigned after the group list has been compiled. The Entity List starts with the M class ETYPE and then adds any instances declared by this ETYPE. These are in turn checked for instance declarations and the process continued until all instances have been created. The Entity List for the Mixer model is shown in Figure 4.14. Parent is the ID number of the entity that created the current entity, type is the ID number of the ETYPE, as stored in the GOL Entity Type List (Figure 4.7), used to create the current entity.

No.	Name	Group	Parent	Type
1	test	1	0	3
2	test.feed.1	2	1	1
3	test.feed.2	3	1	1
4	test.MX	0	1	2
5	test.MX.in.1	2	4	1
6	test.MX.in.2	3	4	1
7	test.MX.out	5	4	1

Figure 4.14: Entity List for Mixer Model

Next, the flowsheet must be connected. The Entity List stores a list of all the entities named in the process. When the units and streams are connected, some of these are found to be the same. These entities are assigned the same group ID number. The Group List stores the list of unique entities in the model and pointers to their variables, equations and initialisation methods. For ease of reference one of the entity names is copied into this list. The Group List for the mixer model is given in Figure 4.15. All other data structures are created from the Group List.

The Variable Set (VS) is created next. Variables are added to the set in the order they

No	Name	VarStart	VarFinish	EqnStart	EqnFinish	IniMethod
1	test	1	1	0	0	1
2	test.feed.1	2	4	1	1	2
3	test.feed.2	5	7	2	2	3
4	test.MX	0	0	3	4	4
5	test.MX.out	8	10	5	5	5

Figure 4.15: Group List for Mixer Model

are declared within the Group List. As variables are added they are assigned values, specifications if given or the default value for the relevant VTYPE. Stat indicates whether the variable is a specification (stat = 1), a user given initial guess (stat = 2) or a default value (stat = 4). Active and integer are flags (1 = true, 0 = false) to indicate whether a variable is active in the current view or is an integer respectively. This is demonstrated in Figure 4.16. The Specification List is set up simultaneously.

No.	Name	Entity	VTYPE	Integer	Lbd	Ubd	value	scale	initial	Stat	Active
1	ncomps	1	1	1	0.0	100.0	2.0	5.0	2.0	1	1
2	F	2	3	0	0.0	100.0	10.0	20.0	10.0	1	1
3	x.1	2	2	0	0.0	1.0	0.3	0.5	0.3	1	1
4	x.2	2	2	0	0.0	1.0	0.5	0.5	0.5	4	1
5	F	3	3	0	0.0	100.0	20.0	20.0	20.0	1	1
6	x.1	3	2	0	0.0	1.0	0.5	0.5	0.5	4	1
7	x.2	3	2	0	0.0	1.0	0.4	0.5	0.4	1	1
8	F	5	3	0	0.0	100.0	15.0	20.0	15.0	4	1
9	x.1	5	2	0	0.0	1.0	0.5	0.5	0.5	4	1
10	x.2	5	2	0	0.0	1.0	0.5	0.5	0.5	4	1

Figure 4.16: Variable Set for Mixer Model

Once the Variable Set exists the Method Calls (MC) and Equation Variable List (EVL) are produced. Each MC table is set up in the same way. Figure 4.17 shows the first three equations set up by the model (sumX for the two feed streams and the mass balance around the mixer). The pointer variable 'iv' in the MC tables points to the location in the EVL one before the location of the first relevant variable. This allows easier referencing in later processes, iv+1 is the first variable, iv+2 the second and so on. Active acts in a similar fashion to the Active flag in the Variable Set, VEnd is used to mark the last variable in the EVL relevant to the current MFC call. The EVL is an integer vector, storing the required variable ID numbers (Value in Figure 4.18). For clarity, two columns have been added: No (position within the vector) and Name

(entity and variable name). MC routines access the necessary variables in the VS through the iv pointer and the EVL data.

No.	Name	Active	VStart (iv)	VEnd
1	sumX	1	0	3
2	sumX	1	3	6
3	massB	1	6	9

Figure 4.17: Subset of Equation Table for Mixer Model

No.	Name	Value
1	test ncomps	1
2	test.feed.1 x.1	3
3	test.feed.1 x.2	4
4	test ncomps	1
5	test.feed.2 x.1	6
6	test.feed.2 x.2	7
7	test.feed.1 F	2
8	test.feed.2 F	5
9	test.MX.out F	8

Figure 4.18: Subset of Equation Variable List for Mixer Model

4.4 Methods

JFMS provides the modelling data and structures required to represent and build equation based models. To actually produce the derivative and residual data needed by the attached solution and optimisation methods external functions are required. The three classes of methods have already been introduced and are equations, objectives and initialisation methods. These all work in essentially the same way, equation and objective functions will generally create derivative and residual data for a given set of variables while an initialisation method will perform some other manipulation on the variable set in order to derive a better starting guess.

Equation and Objective Functions can be considered a refinement of the USMP UTR (see Figure 4.3). Whereas a UTR would produce derivative and residual data for an entire unit type, these functions typically work at an individual equation level and in a much more general and efficient manner. The inclusion of vectors within the variable

set allows a much greater degree of generalisation to exist within a given ETYPE and the equation handling routines must be capable of dealing with this. The level of flexibility required is achieved by the provision of wild cards (*) within entity and variable names in the equation declarations (see page 80) and the conditional and loop facilities provided by writing the functions in a high level programming language such as Fortran90. Efficiency gains are made within the equation by allowing JFMS to specify what is required from the function. Code will exist to produce Jacobian, Hessian and residual data but in many instances only a subset of this data will be required. Solvers for example will often not need Hessian data and only certain values may need to be updated at a given time (see Appendix B for a sample Equation Function).

Initialisation methods are attached to ETYPEs in order to assist the initialisation process. These are covered in detail in Chapter 6. Using the same calling structure as equation and objective functions, these methods can be used to initialise anything from single entities within the model to collections of entities making up complex, interlinked structures.

To all intents and purposes the methods in JFMS are mini packages in their own right, taking a subset of the variable set as input and returning the required information. This results in the modelling package being completely separated from the derivation of these values and gives the user a number of options:

- Values can be obtained using a range of sources: external packages, exact or finite difference methods etc.
- Internal calculations can be performed on the data (ie flash calculations).
- One routine can be used to set up many equations.
- Equations are written in Fortran 90. This allows conditionals to be included in the model; e.g. to avoid indeterminate limits as in the definition:

$$\Delta T_{lm} = \frac{\Delta T_h - \Delta T_c}{\ln \frac{\Delta T_h}{\Delta T_c}} \tag{4.5}$$

  Where $\Delta T_h \rightarrow \Delta T_c$ (Morton, 1996). This keeps the modelling language simple but provides the data handling capabilities of a high level language.

This facility in FMS allows different equation forms to be used to model the behavior of a set of variables over different ranges. Often, physical property correlations are only applicable over a relatively small temperature or pressure range. Just as the equation form can be changed to avoid division by zero it can also be altered to reflect a change in behaviour.

It must be noted however that the equation form should be chosen and fixed the first time it is called. Changing the equation form mid solution will cause convergence problems.

4.4.1 External Packages

External packages are typically physical property data banks but can be other modelling packages or data banks, assuming that an appropriate interface exists to them. Connection to such a package is relatively simple, being an extension of the standard equation function described above, and is outlined below:

- Call JFMS external function with appropriate variable subset.
- Convert JFMS structures to provide correct input to the external package interface.
- Call external package.
- Take results from external package and convert back to JFMS structures.

An example of this, showing connection to a physical property package is shown in Appendix B.

4.4.2 Internal Calculations and Conditionals

Internal calculations and conditionals in the function are methods of introducing complexity to the model without the modelling language having to be capable of expressing

it or the *current* user being faced with the full complexity of the model. Since JFMS is independent of the value generating process it is possible to produce the values in any way the user wishes. This can involve conditionals as illustrated earlier, changing the equation form used to avoid invalid or inappropriate equations based on given criteria (typically division by zero or different equations being used to represent the process in different conditions) or internal calculations.

Internal calculations allow levels of processing to be hidden from the JFMS model; rigorous distillation column models can be run within the equation function in order to accurately model a complex distillation process while the JFMS model appears to the user to be a very simple one, concerned only with the column inputs and outputs. This is frequently the case where non-FMS format models are being used to provide some of the required data.

The example given earlier, using internal calculations to perform flash calculations comes from work done in modelling utility systems. In such systems the streams involved are frequently water or air. In the case of water streams there will be three identifiable forms: liquid, vapour and mixed phase. Use of internal calculations allows a common model structure to be used in all cases with appropriate enthalpy calculations for the phase and an additional flash calculation being performed for the mixed phase case.

4.4.3 Method Library

Methods are accessed through a compiled library linked to the Application Server. This library takes the form of a CASE construct (see Appendix B for a simple example), calling connected external subroutines. Where a large number of functions are to be included it is common to split them into families and use a family prefix. Families are then stored in separate modules, allowing duplicate function names and faster compilation of new or updated code. The exact format used depends on whether the user is inheriting functions from an existing user or not. Entrance to the library is through a function called eqn_handler. After that it is up to the user to specify the

library structure.

4.5 Application Server

The Application Server (AS) provides access to all the attached preprocessing, analysis, solution and optimisation packages. As described in Section 4.3.2, many of the data structures used in the model builder are shared with the application server and so the specifics will not be covered here again. The sole aim of the AS is to pass the model structure and current variable and equation set to an application and to return the results to the model interface on termination of that process. JFMS uses the standard interface to the package wherever possible. As with external package calls in the equation functions, the JFMS data is converted into the appropriate format for the application, the application is run and the output from it converted back into the JFMS format. AS Model Routines provide the relevant links between the application and the JFMS equation handling routines. These routines return Jacobian, Hessian and residual data for the model in question. Figure 4.19 shows a simplified schematic diagram of the process.

Applications currently attached include:

- Linear and Non-Linear Solvers
- SQP Optimisation Routines (Filter)(Fletcher and Leyffer, 1998)
- Physical property library
- Equation analysis tool (Morton and Collingwood, 1998)
- Initialisation tool

4.5.1 Adding Applications

It is not intended or, indeed, desired to provide JFMS interfaces to applications. As stated earlier, applications are run through their own, existing interfaces. This min-

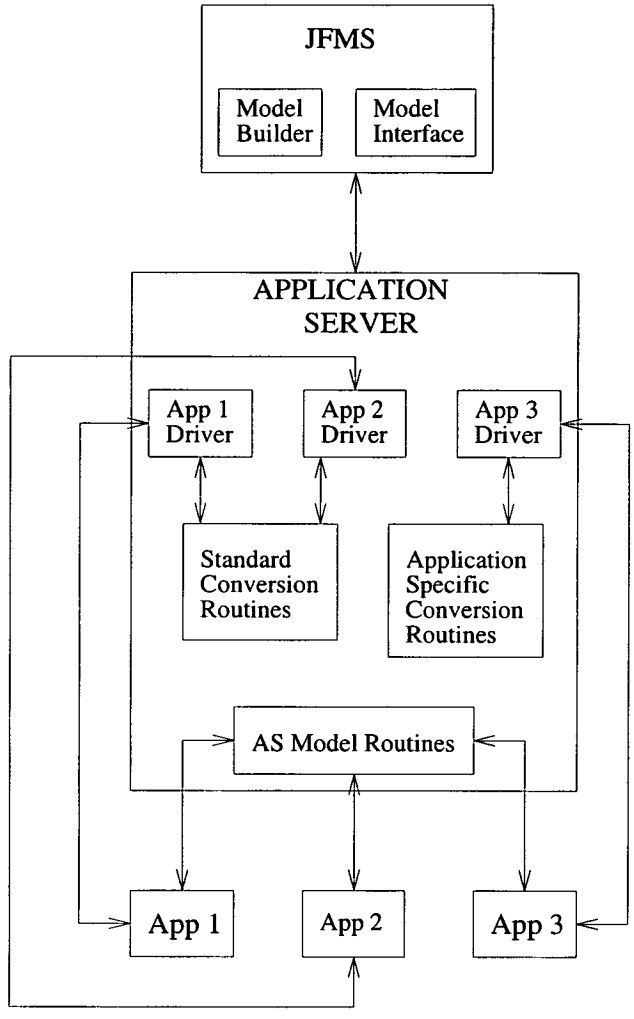

Figure 4.19: Application Server Schematic

imises the amount of work required to add new applications to the system and allows maximum reuse of existing code. JFMS provides routines to calculate residual, Jacobian and Hessian values for a given variable set and to convert between the JFMS formats (typically standard sparse matrix representations) and a number of other, commonly used formats.

Currently, model structure and variable data are transferred from the JFMS core to the Application Server along with a flag informing the AS .which application is to be run on the model. While this works, it means that model structure data is transferred each time an application is run. Control returns to the JFMS GUI when the application terminates and the variable vector is updated. This guarantees that the AS contains

data relevant to the active model in the GUI but is obviously inefficient if the user wishes to perform multiple, consecutive operations on a single model. This could be avoided by adding a verification call to the data transfer to check that the AS model and the GUI active model were the same. Where the models were equivalent there would be no need to transfer the structure data again.

In order to add an application to JFMS the following steps must be followed:

- Add the application to the selection list (menu) in FMSModHandler.java and to the CASE construct in the handleEvent function in the same file.
- Add call to FMS driver for the application to f90exec CASE construct.
- The FMS driver for the application handles the conversion of the FMS format structure and variable data into the format required by the application. The driver then calls the application routine (typically a subroutine call), waits until this routine terminates and then updates the variable set. Control then passes back to the GUI with the return of the updated variable set. Structure data is not returned at the moment but this would be simple to implement and would allow the structure of the problem to change within the application.
- Typically, the application will require access to derivative and residual data and will have routines within it to produce these values. These routines must be rewritten to call the relevant FMS function and appropriate conversions made between the application and FMS data structures. The FMS data structures are accessed through the jac_hess module.

4.5.2 JFMS Equation Handling Routines

The jac_hess module contains five routines of interest:

- fms_objfun(): return current objective function value.
- fms_confun(): update residual data.

- fms_hessian(): update hessian data.
- fms_gradient(): update first derivative data.
- fms_build_jh (cj , ch , cr , co , eval).

The final form is a generalised call to the equation processing routines. It has four logical flags (cj, ch, cr and co) to specify whether first derivative, second derivative, residual or objective function evaluations are required and a real variable to allow access to the objective function evaluation. Three flags influence the behaviour of these routines and are:

- fms_sys%recalc: if true, clears all current derivative etc data and rebuilds. Allows the problem to change structure if necessary or a completely new problem to be started.
- fms_sys%sij: if true, specifications are represented as equations. Otherwise, specs are assumed to be given as tight bounds on the relevant variable or in some other, application specific, method.
- fms_sys%specsFromIni: if true then the temporary specification list is used. This allows temporary specs to be introduced when trying to solve subsets of the model. The global specification list is used if this flag is false.

Advanced Equation Handling Facilities

The flags and routines provide a powerful interface to the equation set. Through use of the correct function, only the desired values will be calculated. This is particularly useful in more advanced solution or optimisation methods where frequently only subsets of the equation data will be required at a given moment in the process and may be recalculated at slightly different points many times in a given iteration. Further performance gains are achieved in the handling of linear equations. Unless the equation data is being reconstructed (as a result of fms_sys%recalc being true) derivative data is not recalculated for linear equations.

The flags, combined with the ability to extract subsets of the variable and equation sets allow the structure of the problem to change if so desired and for different representations of specifications to be handled. Altering the structure of the problem on the fly allows efficient modelling of batch processes, given an appropriate driver, or of synthesis problems where the connections between units and indeed the active units may change as the solution develops.

Chapter 5

Modelling in FMS

5.1 Introduction

FMS provides an object based modelling language that can be easily extended to add new data types to the basic structure. The user constructs models using three data types:

- Variable Type;
- Entity Type;
- Methods.

These have been defined in Chapter 4, pages 47, 48 and 59 respectively. This chapter describes how to build models in FMS and how to extend the modelling language to allow new features. Variable Types and Entity Types are stored in standard text files with the extensions '.vt' and '.et' respectively. The only provision is that ETYPES must be declared before they are used in other ETYPES.

FMS is intended to manipulate a set of variables by applying an attached set of external functions to the variable set. There is no requirement for these functions to represent traditional equations, returning Jacobian and residual data for example, but in the

process systems problems discussed this is usually the case. A brief discussion on equation based modelling is included to illustrate the process of developing such models.

5.2 Equation Based Modelling

Most processes found within a chemical plant can be represented by a set of variables and associated equations. There are a large number of ways that this data can be formulated, some of which are more robust than others, some of which simply do not work and others that are unnecessarily large. This section summarises the general concepts behind equation based modelling.

In simulation problems, the number of variables must be equal to the number of equations plus the number of specifications in the model. When this criteria is met, the problem is classified as 'square' and is potentially solvable. Whether the model can be solved depends on two factors: whether the equation set is correctly formulated and the quality of the initial values for the variable set. Formulation of the equation set is dealt with here, initialisation is covered in chapter 6. In optimisation problems the number of variables exceeds the number of equations and specifications. The extra variables are known as 'free' variables and are manipulated by the optimisation routine in order to satisfy the given model.

Use of an object based modelling language such as FMS allows the user to represent a process in any way that they wish. The modelling approach used to develop the current set of FMS models is described in the following sections.

5.2.1 Modelling Streams

There are two issues when modelling process streams:

- Whether to model them as separate entities within the flowsheet or as part of the unit models;

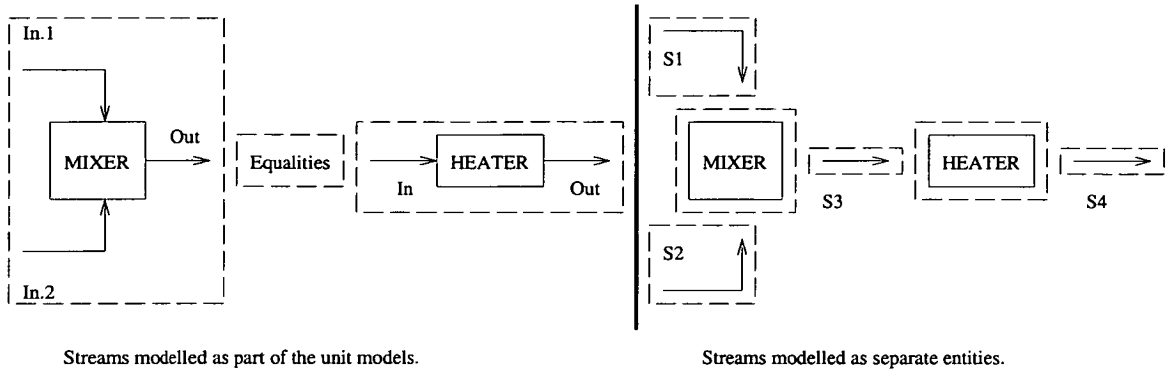

Figure 5.1: Stream Representation

- What variables and equations should the stream model contain.

Streams as Separate Entities

Traditionally, streams have been contained within their associated unit models. This makes referencing the stream variables easier from within the unit and means that the entire model is contained within one routine, making the model easier to understand. However, access to the variables from outside the unit is more complex, relying on a series of pointers stored for each unit. Connecting units will require equality statements of the form found in USMP. There are therefore excess variables and equations and complex addressing issues. There is also no guarantee that any two units will use the same stream model. Models may have different variable and equation sets and could therefore be incompatible.

Representing streams as separate entities in their own right answers most of these questions. Given a system such as FMS the addressing and the compatibility issues are solved and the excess variables removed. This is illustrated in Figure 5.1; items in dashed boxes are entities and each entity will produce a set of variables and equations.

Stream Models

Representing streams as separate entities enforces the use of a standard stream model. There is therefore a need to determine an appropriate set of variables and equations that will hold the necessary information required to represent the stream and to determine any additional information required by the unit models. This set is illustrated in Figure 5.2 for the case of a multicomponent stream.

Data	Variable	Equation
No. of components	Ncomps	
Component IDs	compID(1,Ncomps)	
Component mole fractions	x(1,Ncomps)	$\sum_{i=1}^{Ncomps} x_i - 1.0 = 0.0$
Total Flowrate	F	
Temperature	T	
Pressure	P	
Enthalpy	h	$h - fn(T, P, \bar{x}) = 0.0$

Figure 5.2: Stream model variables and equations

A number of decisions have been made in the selection of this model. It is not a truly minimalist model in that T, P and h exist as variables. Two of these are required, the other could be derived from these and the composition and flow values. However, these are commonly used variables and so including them in the standard model reduces the overhead of having to continually calculate the third value when required and allows easy referencing of the variable by name. The variable set chosen is therefore a balance between size (the number of variables in the model) and ease of access to commonly used values. Mole fractions have been selected instead of component molar flows for several reasons:

- Having mole fraction **and** component molar flows is redundant as one is easily calculated from the other and the total flow variable, F;
- Component molar flows and total flow variables often lead to problems in the formulation of the material balance equation(s), as shown in section 5.2.3;
- Mole fractions are required in most physical property and equilibrium equations.

They are arguably more commonly used than component molar flows which tend to only be needed for component balances.

This structure allows all other likely stream variables such as vapour fraction, K values and liquid/vapour equilibria to be calculated where desired. Streams act as connections between units in the model as they do between units in the process. Given this, models for the units are then created.

5.2.2 Modelling Process Units

Unit models provide the additional variables and equations to determine the effect of the unit on its' inlet and outlet streams. Each model should contain the relevant subset of the following:

- Material balance: Conservation of material across the unit;
- Energy balance: Conservation of energy across the unit;
- Momentum / Pressure relationships: Pressure change across the unit;
- Performance Equations: Model what the unit actually does. A simple example of this would be the split fraction and associated equation in a splitter model.

Derivation of a Mixer Model

The equations for the mixer model in Figure 5.3 are given in Figure 5.4:

The mixer model therefore provides Ncomps + 3 equations, introducing no new variables. Each stream provides 2 equations giving a combined total of Ncomps + 9 equations. Given that each stream has Ncomps + 4 variables the problem contains 3Ncomps + 12 variables in total. In order to convert this into a square problem, 2Ncomps + 3 specifications are needed.

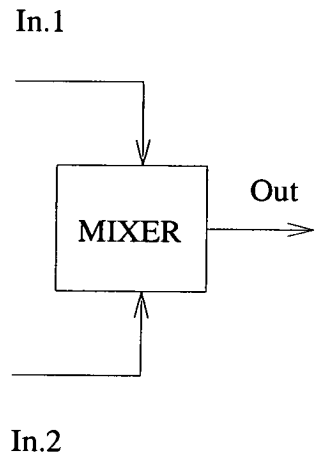

Figure 5.3: Mixer Model

Material Balance

1	Mass Balance	$Out.F = In.1.F + In.2.F$
Ncomps - 1	Comp Balance	$Out.x_i = \frac{In.1.x_i \times In.1.F + In.2.x_i \times In.2.F}{Out.F}$

Pressure

2	Leaving Pressure	$Out.P = In.1.P$ $Out.P = In.2.P$

Performance Equations

none

Heat Balance

1	Enthalpy Balance	$Out.H - (In.1.H + In.2.H) = 0.0$

Figure 5.4: Mixer Model Equations

5.2.3 Modelling Problems

Equation based modelling provides a powerful and flexible approach to modelling processes. However, there are a number of common problems that occur in the formulation and use of such models. These are summarised in the following paragraphs.

Specifications

In order to solve a simulation problem there must be an equal number of variables to the combined number of equations and specifications. The variables that are specified must be chosen carefully in order to allow the problem to converge. A common mistake is to specify all the variables in an equation, resulting in the equation becoming redundant. While a degrees of freedom analysis will suggest that the model is square, there will in fact be a free variable remaining and the model will be under-specified.

This is illustrated below for a system of 4 variables with 2 equations and therefore 2 free variables. Specifying one of a and b and one of c and d produces a square system, specifying a and b or c and d leaves the system insolvable.

$$a + b = 0 \tag{5.1}$$

$$c + d = 0 \tag{5.2}$$

Another problem is that of conflicting specifications. In these cases, specifications have been given that cannot be solved for. Assuming that the flowrates in the mixer model are forced to be positive an example of this would be specifying an outlet flowrate lower than the specified inlet flowrate. In order to converge, the other inlet flowrate would have to be negative. Over-specified problems suffer from both of these problems. Equations will become redundant and, unless the specifications are correct they will contradict each other.

Matrix Singularity

A singular matrix is one in which at some point during the solution process a column or row of the matrix is filled with zeros. This results in the solution method being unable to solve for a particular variable or an equation becoming redundant and, as such, being unable to solve the overall problem. There are two causes of singularity: poor choice of initial values or a badly formulated equation set. Poor choice of initial values cannot totally be avoided but is usually easily sorted, incorrect equation sets can often be more difficult to fix.

Matrix singularity caused by the equation set itself is often a case of equations duplicating the effect of another equation or set of equations. This results in a redundant equation although which equation it is will often not be immediately apparent. This is particularly the case with object based modelling language where a model is often built up from a set of locally declared variables and equations but also creates instances of other models. For example, the mixer model explicitly declares a series of equations to model its' effect on its' inlet and outlet streams. Instances of the stream model are also declared in the mixer model but their contents are not displayed. Without being careful to check the contents of the stream model, the user writing the mixer model may well specify an inappropriate equation set.

Taking the models specified in Figures 5.2 and 5.4 the following should be noted:

- Streams are responsible for ensuring that their component mole fractions sum to 1;
- The mixer model determines the composition of the leaving stream.

An obvious step would be to have the mixer model specify leaving compositions for all components. Since the inlet streams ensure that their mole fractions sum to 1 and the mixer material balance ensures conservation of mass around the unit, the leaving compositions will be correct. However, this leaves a redundant equation in the outlet stream, its' mole fraction summation, which has in effect been duplicated by the mixer model solving for all the leaving components. The solution to this is to have the mixer solve for all but one of the leaving compositions (Ncomps -1) and allow the stream equation to solve for the remaining variable. The problem with this type of singularity is that the equation set will seem logical until the duplication is found, a process which can take considerable time in complex models. An equation analyser (Morton and Collingwood, 1998) has been developed which flags equations or variables likely to be the cause of the singularity.

Equation Representation

Once a suitable equation set has been decided on, how the equations themselves are formulated has an effect on the solution process, particularly in terms of robustness and solution time. The two most common issues are division operations within the function and, given the equation structure of FMS, constant re-evaluation of a given term and these are outlined below.

Division operations should be avoided wherever possible in the equation set. Functions containing division operations are subject to two main problems. Firstly, should the denominator evaluate to zero the function will return an infinite result. Secondly, where the denominator is extremely small the returned value will be subject to large rounding errors. Most equations can be rewritten to avoid division, as illustrated by equations (5.3) and (5.4).

$$a - \frac{b}{c} = 0 \tag{5.3}$$

$$(a \times c) - b = 0 \tag{5.4}$$

Where a term within a model is being constantly re-evaluated inside different equations elsewhere in the flowsheet it leads to a large number of unnecessary calculations. Often this can be resolved by explicitly including the term as a variable in the model although the computational savings must be weighed against the added size and complexity of the new model. Where the term is only needed for a particular equation it should be evaluated within that equation routine. Efficiency gains can be made by prior evaluation of as many of the common terms within the equation routine in general, resulting in fewer calculations being performed overall. These variables can either be included in the global variable set or declared locally within the equation method.

5.3 Building Models in FMS

In order to demonstrate the steps required in building a model in FMS, this section contains an outline of the development of a simple distillation column model. This

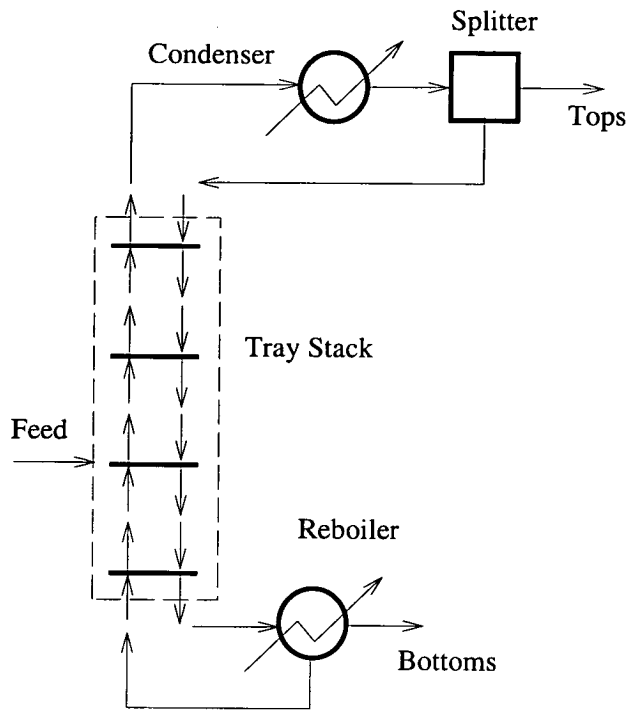

Figure 5.5: Components of a Distillation Column Model

model uses most of the features of the language and is outlined in Figure 5.5.

The column model itself is constructed from a number of other models, these being:

- Liquid and Vapour Streams;
- Condenser;
- Partial Reboiler;
- Distillation Tray;
- Feed Tray (refinement of Distillation Tray);
- Splitter;
- Tray Stack.

5.3.1 The Basic Stream Model

A basic stream model will contain flowrate (F) and mole fraction (x) data. Typically, the number of components (ncomps) and some component identification (cid) will also be required. In order to complete the model a mole fraction summation would be added. This model could be represented in FMS as shown below:

```
ETYPE   basicStream
VARS
        ncomps      integer
        cid         integer     ncomps
        x           fraction    ncomps
        F           flowrate
EQNS
        sumX
                    .           ncomps
                    .           x.*
        ;
END
```

Figure 5.6: ETYPE basicStream

This declares a new ETYPE called basicStream which contains an integer variable called ncomps, an integer vector of size ncomps called cid, a vector of type fraction of size ncomps called x and a value of type flowrate called F. Variables are declared using the structure:

```
VARS
        var name    var type    [size of vector OR vector start value]    [vector end value]
        etc
```

Figure 5.7: Variable Declaration

The variable type has been described on page 4.3.2 and is a way of assigning initial values and bounds data to the variable set in a way that reflects what the variable is intended to represent. An example of this is the flowrate type as shown:

VTYPE *flowrate*
UBOUND *1000*
LBOUND *0*
SCALE *50*
VALUE *100*
END

Variables are assumed to be scalar unless one or more of the optional slots are filled. These are indicated in Figure 5.7 as the entries surrounded by square brackets ([ ]). These values can either be integers or references to integer variables such as ncomps in Figure 5.6. If only one slot is given the vector is assumed to start at position 1 and run to the value given in the slot (ie x.1 to x.[ncomps]). Where both values are given, the values are starting and finishing positions for the vector. This allows vectors to start at values other than 1 should that be required. A particular value within a vector is referenced by concatenating the variable name and the integer position, separated by a period (.). For example, x.3 is element 3 of a vector x. When referring to a vector, the full vector can be indicated by use of the wild-card symbol, *. In this example, the variable name takes the form x.*, which refers to x.1, x.2,..., x.[ncomps].

The basicStream ETYPE also declares an equation called sumX which takes as arguments the local variable ncomps and the local vector x. These are specified as local by replacing the *entity name* slot with a period (.). Equations are declared as shown in Figure 5.8:

```
EQNS
        eqn name
                    entity name    var name
                    etc
        ;
        etc
```

Figure 5.8: Equation Declaration

Section Variables

The model given in Figure 5.6 is sufficient to describe a basic process stream. From a modelling perspective however it is rather clumsy, storing ncomps and cid data for every stream in the process results in a large number of effectively redundant variables in the flowsheet, unnecessarily increasing its size. FMS introduces the concept of *section variables* to avoid this. A section variable is one that applies to all entities (units, streams etc) within a given subsection of the model. Each entity stores the identities of its ancestors (the entities that created it or its parents and so on). Where a value for a section variable is required, the entity will search back up its family tree until it finds an entity holding the required data. Using section variables saves space and is a more realistic reflection of what is happening in the model. Section variables are identified by a hat symbol (^). The model is therefore changed as shown in Figure 5.9.

```
ETYPE   basicStream
VARS
        x           fraction    ^ncomps
        F           flowrate
EQNS
        sumX
                    ^           ncomps
                    .           x.*
        ;
END
```

Figure 5.9: ETYPE basicStream, using Section Variables

Usage differs between the variable and equation declarations. In the variable declaration the variable name is preceded by the hat symbol; in the case of the equation declaration the hat is given as the entity name.

5.3.2 The Thermodynamic Stream Model

For a column model we require the stream models to contain thermodynamic data as well. In particular, we wish to add enthalpy (h), pressure (P) and temperature (T) to

the model. Rather than rewriting the model from scratch we can reuse the code from the basic stream model using inheritance and add the required elements to this. The new model is shown in Figure 5.10.

```
ETYPE       thermStream
CONTAINS
            basicStream
VARS
            h           enthalpy
            T           temperature
            P           pressure
END
```

Figure 5.10: ETYPE thermStream, showing inheritance

Inheritance copies all the data from the parent or parents to the new class. Any additional information required is then specified as normal. This is declared using the CONTAINS keyword in the form, as illustrated in Figure 5.11.

```
CONTAINS
            parent name
            etc
```

Figure 5.11: Inheritance Declaration

In order to complete the stream model an equation must be given to calculate enthalpy. Using inheritance, the thermStream model is developed into liqStream and vapStream, each of which inherits all the data of thermStream and thus basicStream and add equations to calculate liquid and vapour enthalpies respectively.

5.3.3 The Tray Model

A column can be viewed as a collection of distillation trays connected together. A basic tray model is developed using the methods described above and is shown diagrammatically in Figure 5.12. In order to maximise the reusability of the code no mass

or enthalpy balance equations are included in the basic model. The stream models used are as described earlier and each tray stores a local temperature and pressure along with K values to calculate the vapour/liquid equilibrium. Basic tray equations exist to:

- calculate K values
- calculate Vapour / liquid equilibrium
- set leaving stream pressures and temperatures

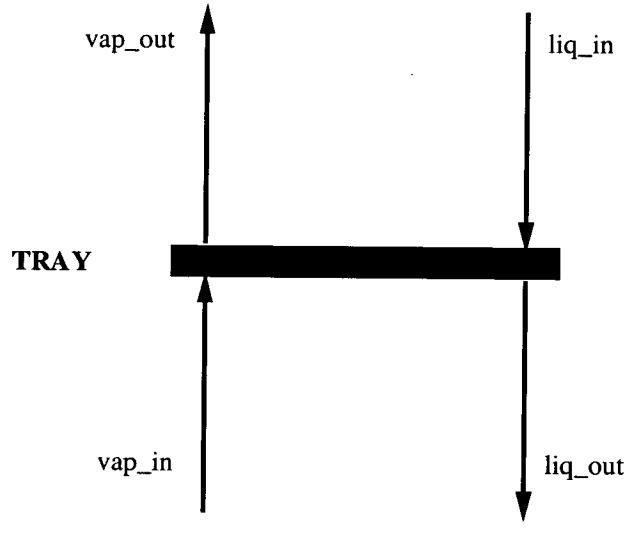

Figure 5.12: The Basic Tray

A simplified FMS ETYPE for the basic tray is given in Figure 5.13. Each tray declares instances of vapStream and liqStream to represent its connections. This is done using the INSTANCE construct as given in Figure 5.14.

In this case, each tray declares four streams; vap_in, vap_out, liq_in and liq_out. As with the variable declaration, a vector of instances can be declared. This is done by specifying the optional vector size shown in Figure 5.14, the value being either an integer number or a reference to an integer variable. This is illustrated in the ETYPE for the tray stack shown in Figure 5.16 as is the actual connection of the trays. Using inheritance, ETYPES for internal trays and trays with feed and/or side streams

```
ETYPE     basicTray
VARS
          K          real          ^ncomps
          T          temperature
          P          pressure
INSTANCE
          vapStream
                     vap_in
                     vap_out
          ;
          liqStream
                     liq_in
                     liq_out
          ;
EQNS
          yKx
                     ^             ncomps
                     vap_out       x.*
                     .             K.*
                     liq_out       x.*
          ;
          etc
END
```

Figure 5.13: ETYPE basicTray, showing instance

```
INSTANCE
          entity type
                     instance name   [vector size]
                     etc
          ;
          etc
```

Figure 5.14: Instance Declaration

are easily developed by adding relevant mass and enthalpy balance equations and the necessary instances.

5.3.4 The Tray Stack Model

The Tray Stack model is then simply a collection of internal trays (ETYPE intTray) connected together. In order to connect entities, the CONNECT construct is used. This is discussed in chapter 4, page 53 and is outlined in Figure 5.15. If the entities are of different type the entity highest in the Entity List (see page 52) is used to determine the final type of the group.

Typically, the connection being made will be between two entities such as an outlet stream from a mixer and the inlet to a reactor. Entity 1 and entity 2 in Figure 5.15 are therefore single instances. In the tray stack model however there is an unknown number of trays to be connected. In order to handle this, FMS allows evaluation of expressions within the entity name and list handling facilities. The first connection being declared in Figure 5.16 is:

```
CONNECT
                tray.[*].vap_out    tray.[* + 1.0].vap_in
```

When parsing the entity names, elements contained within square brackets ([ ]) are to be evaluated. The expression can contain integers (expressed as reals), reference to integer variables (entity and variable name) or wild card symbols (*). These can be linked with either + or - symbols, each element being separated by a space. In the case above, tray.[*].vap_out finds all the vap_out streams in the stack (tray.[1].vap_out to tray.[ntrays].vap_out) and stores their positions. The second expression, tray.[* + 1.0].vap_in, does the same for the vap_in streams. Removing the '+ 1.0' from the second expression would result in tray.[i].vap_out being connected to tray.[i].vap_in for i=1,ntrays. This is not the desired connection however and the '+ 1.0' gives the relevant offset (tray.[i].vap_out connected to tray.[i + 1].vap_in, i=1,ntrays).

```
CONNECT
            entity 1    entity 2
            etc
```

Figure 5.15: CONNECT Declaration

```
ETYPE       trayStack
VARS
            ntrays              integer
INSTANCE
            intTray
                                tray                    ntrays
            ;
CONNECT
            tray.[*].vap_out    tray.[* + 1.0].vap_in
            tray.[*].liq_out    tray.[* - 1.0].liq_in
END
```

Figure 5.16: ETYPE trayStack, showing connect

5.3.5 The Column Model

Models for the remaining ETYPES are produced in a similar fashion. The final column model is defined in Figure 5.17. Variables are included to specify feed stage position and the pressure drop across each stage. The addition of a pressure drop equation completes the generic description of a distillation column.

When connecting the feed tray streams to the relevant streams in the column stack it is necessary to use the offset list form to get the correct connectivity. This is because the feed tray entity has a feed stream instance in front of the standard tray streams whereas the stack tray does not.

```
ETYPE     column
VARS
          f_pos          real
          cdp            real
INSTANCE
          trayStack
                         stack
          ;
          feedTray
                         fTray
          ;
          condenser
                         cond
          ;
          partialReboiler
                         reb
          ;
          splitter
                         split
          ;
CONNECT
          fTray          stack.tray.[. f_pos]
          fTray.[*]      stack.tray.[. f_pos].[* - 1.0]
          cond.vap_in    stack.tray.[stack ntrays].vap_out
          reb.vap_out    stack.tray.1.vap_in
          reb.liq_in     stack.tray.1.liq_out
          split.liq_in   cond.liq_out
          split.liq_out.1 stack.tray.[stack ntrays].liq_in

EQNS
          col_dp
                         stack                    ntrays
                         stack.tray.*             P
                         .                        cdp
          ;
END
```

Figure 5.17: ETYPE column

5.3.6 Using the Column Model

As the column model stands it is a generic description of a column. In order to use the model an ETYPE must be created to represent the flowsheet that the specific instance

of the column model belongs to. This introduces two more constructs to the modelling language: FIXED and ASSIGN. These are used to give specifications and initial values respectively for all slots in the variable class (value, upper and lower bounds and scale factor). The following form is used, KEYWORD being replaced by either FIXED or ASSIGN as appropriate. Variables can be specified or assigned initial values in any ETYPE although in order to maximise the flexibility of the code it is suggested that this is done in the flowsheet ETYPE. The structure used to fix or assign values is given in Figure 5.18. A cut down version of the flowsheet ETYPE is shown in Figure 5.19.

```
KEYWORD
        entity name   var name   slot   value
        etc
```

Figure 5.18: Specification / Assignment Declaration

```
ETYPE       flowsheet
VARS
            ncomps              integer
            cid                 integer   ncomps
INSTANCE
            column
                                C101
            ;
FIXED
            .                   ncomps    value    3.0
            .                   cid.1     value    2.0
            C101.stack          ntrays    value    10.0
            C101.fTray.feed     F         value    100.0
ASSIGN
            C101.split.liq_out.2  F       lbound   45.0
END
```

Figure 5.19: ETYPE flowsheet

5.3.7 Other Basic Language Features

There are two other features of the language not included in the column model described above; objective function and initialisation method calls. These were introduced in Chapter 4, page 59 and take the same form as the EQN construct outlined in Figure 5.8, replacing the keyword EQN with OBJ or INIM respectively.

Objective functions can be declared within any ETYPE. Where multiple objective functions are declared within a flowsheet the function belonging to the entity highest in the Entity List will be used.

Initialisation Methods are used to produce better starting values for the model. Usually this will require either a user written initialisation routine or a call to an external library or application. Where multiple initialisation methods exist within the inheritance tree of a given ETYPE, the most refined ETYPEs method is used.

The structure of an ETYPE data file is shown in Figure 5.20.

5.4 Object Orientated Modelling

While FMS is not truly an object orientated language, many of the concepts used in its development come from the Object Orientated (OO) methodology. Main features of an object orientated system are:

- Classes - data structure with attached methods to manipulate and access that data;
- Instances - a specific member of a class;
- Inheritance - subclasses can inherit features from its superclass;
- Encapsulation - access to parts of the class can be restricted;

- Overloading - the ability to use the same name for a function with different arguments;
- Polymorphism - the ability to handle types with common features.

5.4.1 Classes and Instances

ETYPES and Entities are the FMS equivalent of OO classes and instances. External functions such as the equation routines and initialisation method calls provide the methods to handle the data in the object.

5.4.2 Inheritance

One of the most important features of an object based language is the idea of inheritance. In order to use this feature to its full extent, care must be taken in developing the ETYPES used from the start. The column model is a good example of careful design allowing maximum reuse of code, this being particularly apparent in the design of the distillation tray and stream models.

When developing a set of related ETYPES it is important to isolate the features that they have in common. In the example of the distillation tray, all trays, regardless of side streams, feeds or other conditions will need to store pressure, temperature and K values for the tray and be able to set leaving stream compositions, temperatures and pressures. This therefore forms the root ETYPE for the tray group, the other ETYPES being produced by extending this one. Many ETYPES produced are therefore not capable of being used in a flowsheet. While this at first glance appears to cause an unnecessarily large library of ETYPES to be required, the savings made by reusing the code, both in terms of time taken to write the models and in the time taken to debug errors justify this.

Wherever possible, the additions made at each stage should be as small as possible and unless absolutely required no specifications should be introduced in ETYPES that are

to be further developed. This allows maximum reusability and flexibility. If sufficient care is taken in isolating the common features of the group at each generation then there is no need to allow selective inheritance. Selective inheritance allows the user to choose which bits of the parent class are copied into the child directly and which are to be altered. This can be useful if altering existing ETYPES that have been constructed in a less flexible manner but this should be left to the individual.

5.4.3 Polymorphism, Encapsulation and Overloading

At the level that classes interact within FMS from a modelling perspective, there is a degree of polymorphism displayed. As surrounding entities are only concerned with the external connections from a given entity, any subclass that extends the original ETYPE used can be used in its place. Encapsulation and overloading do not occur within the FMS language but are used within the underlying code.

5.5 Extending the Language

An important feature of a modelling language is the ability for the user to add new functionality to it when required. FMS has been developed in a modular fashion in a commonly used language. This section provides an overview of the steps required to add new data structures to the language; it should be noted that new models are written in the model definition language and do not require this level of coding. Extending FMS is simplified by the fact that it is solely intended to read data from a text file, convert this into a simple representation and pass this on to the application server. The main steps are outlined below and covered in more detail afterwards, relevant filenames are given in italics.

- ETYPES are stored in Java objects (*EntT.java*) and new data structures must be added here;

- At run time, a GOL (Generic Object Library) object is created (*GOL.java*). This object reads the text file and stores it in the relevant EntT;
- When a particular flowsheet is built it is stored as a NGOL (Non-Generic Object Library) object (*NGOL.java*) by the method createNGOL (*ModelBox.java*). This completes the transfer of data from the original text file to JFMS;
- The final step is the transfer of data to the application server. This is performed by the Java method sendData (*FMSModHandler.java*) and the F90 routines *addreal*, *addint* and *addstring*. The data is transferred into the F90 variables in *global.f90*. Depending on the nature of the data, it may or may not be necessary to alter these structures.

5.5.1 Entity Type Storage

JFMS Entity Types are stored in a vector in the GOL. Each element in this vector is an object of class EntT and therefore this class is the starting point for any additions to the language. Any new data structures that the user wishes to include in their models must be reflected in this class along with the methods required to access and add data to them. The GOL also contains the method required to parse the input file. This method (readETF) reads the data from the input file line by line, working on a flag based parser. The following procedure is followed:

- Check whether line is input code or comment, ignore if the latter;
- If line contains a keyword then set relevant flag and read next line. Otherwise, process under existing flags;
- Parse data according to flags and add to relevant structure in the current ETYPE.

5.5.2 Model Storage

JFMS models are stored in a vector of class NGOL - each model being a complete flowsheet with associated variables and methods. This class must hold all the data

relating to the model and therefore all the data stored in a generic form in the ETYPES must be transferred to the specific instance of the model. The model is created by the createNGOL method and this must be adapted to reflect any changes in the basic ETYPE structure. Once the data is stored in the NGOL what is required depends on what the data is intended to do. Most data will need to be transferred to the Fortran90 based application server and this is outlined below.

5.5.3 Data Transfer to Application Server

The Java method sendData is responsible for sending the data to the application server. In order to minimise the complexity of the transfer procedure the Java structures are broken down into their component real, integer and character vectors. Due to the differences in string handling between C and Fortran90 the string arrays are further broken down and transmitted as a series of separate strings. The functions addreal, addint and addstring receive the data and assign it to the appropriate Fortran90 structure.

```
ETYPE       type name
CONTAINS
            parent 1
            etc
VARS
            variable name   variable type    [vector start]   [vector end]
            etc
EQNS
            equation name
                            *entity name     *variable name
                            etc
            ;
            etc
INIM
            equation name
                            *entity name     *variable name
                            etc
            ;
INSTANCE
            entity type
                            instance name    [size]
                            etc
            ;
            etc
FIXED
            *entity name    *variable name   slot             {value}
            etc
ASSIGN
            *entity name    *variable name   slot             {value}
            etc
CONNECT
            *entity name    *entity name
            etc
END
```

- Entries in capitals are reserved key words;
- Entries preceded by a star may contain wild cards or expressions requiring evaluation;
- Entries in [ ] are integer values or refer to local integer variables by name, those in { } are real values.

Figure 5.20: Generalised Structure of an ETYPE Data File

Chapter 6

Initialisation Methods

6.1 Introduction

State of the art models of chemical plants frequently involve upwards of 100,000 variables and equations. The resulting system is typically highly nonlinear and sensitive to small changes within the operating parameters. This results in a sharp increase in the difficulty of solution, both in terms of robustness, whether a solution is obtainable or not, and in the time taken to produce the solution if one is achievable.

In order to maximise the chance of obtaining a converged solution some form of preprocessing of the problem usually occurs. This can take several forms, as outlined below:

- User given initial values for all variables;
- 'Hot starts' - solving subsets of the problem and gradually combining these to produce the complete model;
- Matrix Reordering;

Within the chemical industry the user is generally aware of the approximate solution and has a good knowledge of the process being modelled. A package called REFORM

(Amarger et al., 1992) was developed which aimed to capture this qualitative user knowledge and reformulate GAMS models on the basis of the resulting knowledge base. The reformulated model was found to be more robust and the efficiency of the optimisation algorithm is increased.

To illustrate, 'Process systems' models are notoriously sensitive to both energy and material flow directions and often have well known behavioral patterns. An example of the latter is the temperature and pressure profile in a distillation column. We believe that while user participation is essential within the initialisation process a large amount of the basics can be automated. The user should be left to provide the essential direction, rather than actual values wherever possible.

The method being proposed combines the mathematical and user driven approaches. The flowsheet is broken down into its component units and streams and a starting unit or stream for the initialisation identified. If this point is well specified or relatively simple to solve then an NLAE or optimisation method can be used. This is easily achieved in FMS due to the ability to extract sections of the equation and variable sets. Where the component is more complex, the generalised code for the unit will include an initialisation application to assist the user.

The underlying structures and code to support this approach have been developed and are present within FMS. This chapter presents a number of possible initialisation methods and a potential implementation within FMS. It is hoped that this work will be continued with a view to validating the method proposed.

6.2 Existing Techniques to Aid Convergence

6.2.1 User Given Values

Newton-based methods for solving highly non-linear problems are very sensitive to the initial values given. A poor set of initial values will often prevent the solution from converging; depending on the complexity and degree of non-linearity there is no need for

the model to be particularly large. Therefore, this level of initialisation will frequently be used on a unit by unit basis. There are several common options for initialising a variable set and these are outlined below.

- Bounds based default value. The default value is set at the mid-point of the variables range, as defined by the upper and lower bounds. Can be off by several orders of magnitude from the final value. This is particularly the case for variables with no real physical limit where the allowable range must be almost infinite, e.g. heat exchanger areas.
- User given default value. Still very general by its very nature but probably more accurate than the bounds based estimate. Usually applies across an entire flowsheet, i.e. all flows are set to a default of 100 kmol/h. This has the advantage that the bounds can be kept relatively wide without affecting the initialisation.
- User given local value. More specific again. This will generally apply to a single variable or possibly a sub-section of the flowsheet (i.e. all flows within the separation section are initialised at 75 kmol/h). Often it is at this level that the user must perform the bulk of the initialisation, frequently on a variable by variable basis.

As with most current modelling packages, FMS works at the user given default and local value levels. This allows rapid allocation of non-vital default values with the ability to focus on and alter solution critical values.

6.2.2 Hot Starts

Process system models typically represent large sections of a chemical plant, if not the plant in its entirety. While a simple initialisation technique as described above is often sufficient for very small or simple problems it will often be defeated by larger, more complex ones. The traditional method in this case has been to solve subsets of the model, gradually adding new units to the model and building the desired model

in a piecewise fashion. This is often referred to as 'hot starting' a model - a large proportion of the model is already converged and only one or two units need to be solved for. Depending on the complexity of the new units it may be necessary to provide good initial values for their variables as well.

6.2.3 Matrix Reordering and Decomposition

It is sometimes possible to improve the robustness of a problem or to reduce solution times by reordering the equation matrix and decomposing the resulting matrix into blocks of equations and variables that can be solved independently. Traditional modelling techniques often result in a relatively random matrix and this can lead to problems in the Gaussian Elimination (GE) step. The most common of these is a result of a poorly formulated equation set known as singularity. In this instance, a row or column of the matrix is filled with zeros during the GE step and, as a result, it becomes impossible to calculate a value for the variable on which the row would otherwise pivot. Assuming that the model is well formulated, a random matrix often produces a large degree of fill-in during the GE step. This reduces the degree of sparsity within the matrix resulting in a reduction in the efficiency and robustness of the sparse solution methods. In the case of a random matrix, reordering is desirable in order to retain the sparsity of the matrix.

There are two forms of decomposition: equation based and structural. Purely equation based methods (Stadtherr and Wood, 1984) involve decomposing the model into small subsets of equations. These subsets are either linear in their variables or can be reduced to solvable forms. Such methods work well where the model can be broken up into relatively small or easily solvable subsets. Frequently however, there are one or more large and complex irreducible blocks formed which retain much, if not all, of the solution's complexity. In these cases equation based decomposition methods are often of little or no assistance.

The second form of decomposition is based upon the structure of the process that the model represents. One of the features of an object based modelling language such as

ASCEND (or FMS) is that the equation matrix is typically created on a unit by unit basis. This results in distinguishable blocks of variables and equations representing each unit and stream (Abbott et al., 1997). The work performed by Westerberg examined the reduction in solution times possible by exploiting the structure of such equation and variable sets. At the limit, such an approach tends towards a sequential modular solution method, the equations for each unit being solved simultaneously. This approach was initially implemented in ASCEND-II (Locke and Westerberg, 1983). This however does not answer the problem of a large complex unit or section of plant which cannot be solved for even as an isolated model.

It is proposed to add an initialisation method to the model description (EType in FMS) in order to capture the users knowledge of the unit in question and to automate the initialisation process. This should combine the benefits of the user's process knowledge (Amarger et al., 1992) and the structure of the model (Abbott et al., 1997).

6.3 Project Outline

The project proposes an automated initialisation package for use with the Flexible Modelling System (FMS) developed by the group. Two FMS related terms used within this report are Entity and EType. An entity is an identifiable object within the process such as a stream, reactor or distillation column with an associated set of equations and variables. An EType is a generalised description of an entity and is used to construct the model.

The proposed solution imitates the decomposition of the flowsheet into manageable sub-components as typically performed by the user. This produces a list of entities, each of which is assigned a score based on how well described it is in the model formulation. Initialisation then starts at the best known entity which is then removed from the list and the remaining entities scores reevaluated. The process of initialising and re-scoring continues until the entity list is empty.

6.4 FMS_INIT

FMS_INIT is the package developed to assist with the initialisation process. It contains a number of routines and data structures, each of which can communicate with the core FMS routines and through these any other attached packages. These are:

- An active entity list;
- An entity scoring routine;
- A library of initialisation routines .

6.4.1 Active Entity List (AEL)

An active entity is an entity which has not yet been initialised and is eligible for initialisation. The latter requirement removes entities representing objects such as distillation column trays or other unit internals from the list. These are removed from the AEL to allow, for example, the column to be initialised as a complete unit. In order to allow user written routines to initialise at a larger scale such as providing an initialisation routine for an entire plant section in order to handle a recycle, entities can be removed from the AEL by the user. Entities are removed from the AEL when they have been initialised and the process finishes when the list is empty.

6.4.2 Scoring Routines

In order to decide which entity to start the initialisation process at, the package requires some method of measuring how well 'known' an entity is. This measure is based on the total number of variables contained within the entity and then an analysis of the state of each of those variables.

The states used within the package, in increasing order of confidence are:

- User given default value;
- User given local value;
- Value derived from FMS_INIT;
- Warm start derived from another solution;
- Specification.

Each state has a user given weight (W1 to W5) and each entity returns the numbers of variables of each state (N1 to N5) as well as the total number of variables (NT).

The scoring routine produces a confidence ratio for the entity. This is the ratio between the actual score for the entity (S) and the highest possible score attainable. For the purposes of the package this is taken as being the score obtained if all the variables were specifications, i.e. NT * W4.

$$S = N1W1 + N2W2 + N3W3 + N4W4 + N5W5 \quad (6.1)$$

$$CR = S/(NTW5) \quad (6.2)$$

Once CRs have been obtained for all entities on the AEL, initialisation starts with the entity with the highest CR. Where two or more entities have the same CR the entity with the fewest variables is selected, unless the user wishes to override the choice. This maximises the chance that a higher fraction of the specified or otherwise non-default values are critical values and therefore increases the validity of the derived values.

Some measure of ease of solution and likely accuracy of the solution is required in order to influence the choice of entity to initialise. Scoring as described above provides one such measure but others should be investigated.

6.5 Initialisation Methods

There are two methods used to initialise entities in FMS_INIT: sub-problem extraction and solution or user written application. Where the entity in question is small or

trivial to solve the user need not be involved in the initialisation process. In this case the former method is used. For complex structures (e.g. distillation columns) which require a large amount of initialisation the relevant EType will provide a link to an initialisation routine. This routine can perform a variety of tasks intended to assist the user. These can include short cut models of the unit, analysis of the results and iterative procedures for recycles. Where these methods fail or are not applicable a genetic algorithm approach has been implemented.

6.5.1 Sub-problem Extraction and Solution

The ability to extract sub-sets of the variable and equation sets allows extraction of the data for a particular unit. The application can use this to solve (or initialise) the model in a pseudo sequential modular manner, as is typically done manually.

FMS extracts the equation and variable set for the given entity from the global lists and passes these to FMS_INIT. The specific function used is tailored for the application. JFMS provides a standard function (GrpVESubset in module jac_hess) to perform this function on a entity specific basis.

In the simplest case the resulting system is square (number of equations is equal to the sum of the number of variables and specifications) and the system can be solved using one of the standard solvers. Assuming a solution is reached, the status of all non-specification variables is upgraded to that of an FMS_INIT variable and the entity removed from the AEL. Given the complexity of entities that will be initialised in this manner, being unable to reach a converged solution will almost definitely point to one of the following problems:

- Conflicting specifications;
- Too tight bounds;
- Invalid equation set.

At that point the user is able to examine the variable and equation sets and alter any values that are causing problems. This process is repeated until a solution is obtained or the user exits the application.

Where the problem is non-square the user must provide temporary specifications to allow solution (unless an optimisation method is used). These only take the status of a specification (state 4) for any calculations on the relevant entity, the rest of the process seeing them as user given local values (state 2) should the initialisation not produce output that the user accepts. FMS_INIT stores a list of these temporary specifications, allowing subsequent analysis by the user of decisions made during the initialisation in case they need to be changed.

6.5.2 User Written Initialisation Routines

As with the previous method, FMS extracts the relevant equation and variable set and passes these to FMS_INIT. The EType for the entity contains a slot for initialisation method and FMS_INIT passes the data to this method. Typically, this is an F90 application, providing a range of short cut methods, analysis tools and access back to the core FMS routines and therefore any attached physical property packages or solvers.

It is important to note however that since the routine is written in a standard programming language there is no real limit on where the initial values are obtained from. Links exist between Fortran90 and a large number of modelling packages and standard physical property databanks and these can be called from within the routine. Hot starts can also be included from previously solved problems, one possibility being to build a library of 'standard' column profiles and to extrapolate a likely column profile for the specific column from the most similar process in the library. As with FMS, the intention has been to leave the application as general as possible. No restrictions are placed on how or where the user gets the values.

As mentioned earlier, the entity in question need not be a single unit or stream. By

removing sub-components of, for example, the separation section of a plant from the AEL the initialisation could be performed at the level of the separation section. While the routine to do this would be more complicated than one for a single unit it has the advantage that it often promotes consistency within a larger area of the model and allows recycles to be included within the initialisation. A cautionary point that should be made is that the initialisation is intended to be a pre-processing step and not to produce a converged solution. This is especially important for sections or entities lower down in the AEL where their input values are based on many previous initialisations and therefore are only an approximation of their converged values. The aim here is to do as little work as possible before passing the problem to a standard solver and not to replace it.

6.6 Optimisation Based Initialisation Methods

Another possibility is to use an optimisation based method to perform the initialisation. Frequently this will involve the minimisation of a sum of absolute values of residuals, as illustrated in eqn 6.3 or a variation thereof.

$$\begin{aligned} \min . ConV \quad &= \quad \sum_{i=1}^{neq} |c_i(\mathbf{x})| \qquad (6.3) \\ \text{subject to :} & \\ \mathbf{xlb_i} \leq \mathbf{x_i} \leq \mathbf{xub_i} & \end{aligned}$$

Since optimisation methods can handle nonsquare problems, the user will not have to supply temporary specifications for the subproblem as with the methods described earlier. While this makes the system easier for the user, it removes the advantage of user knowledge to influence the initialisation. User knowledge could be represented by use of tighter bounds on the variables.

Figure 6.1 illustrates this problem; where only the input is specified there are many solutions to the model, ranging from zero flow in stream B to zero flow in stream C. An optimisation based method would produce a valid answer but lacking the input from the user this may range from useful to useless with little to tell which. Suitable selection

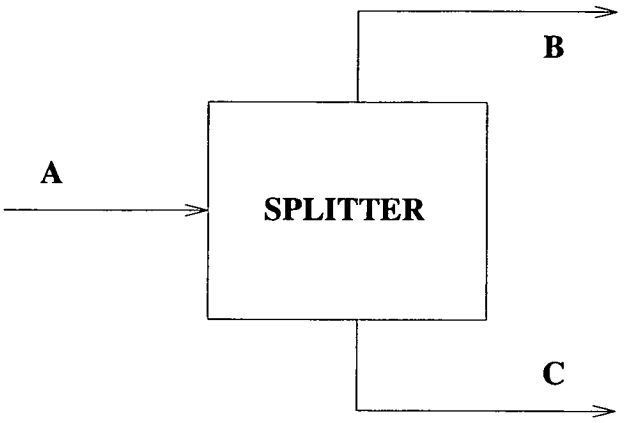

Figure 6.1: Simple Mixer

of termination criteria, in particular the acceptable values of ConV, are critical. Too low a value results in a converged solution being produced and is therefore too costly in terms of time; too high a value terminates the optimisation before the values are even close to where they should be. Such methods are therefore useful only where:

- The user has no idea of what the system will do and any converged (or nearly converged) set of values are better than none at all;
- The sub-problem as it stands is numerically singular and no progress can be made using a deterministic approach. Partial or total optimisation using a stochastic optimisation method provides a possible way forward.

Initial research focused on the use of stochastic methods as deterministic ones are in effect used as NLAE solvers once sufficient temporary specifications have been given to produce a sensible result. It would be possible to allow the user to set tighter bounds on certain values rather than specifications and then run such a method in order to obtain an initial state. This was not investigated but would allow a degree of user knowledge to influence the result.

6.6.1 Genetic Algorithms

Genetic algorithms (Michalewicz, 1994) are a subclass of the stochastic optimisation methods available. They are based on the ideas of evolution and this section is intended to give a brief overview of the theory behind them. Evolution suggests that a species will change over time in response to external stimuli. Fitter members of the species, more suited to the environment, will be more likely to breed and pass on their genes to the next generation. The child gains a mixture of the parents' genes, hopefully leading to it being more adapted to the environment. Over time, members of the species die, either through old age or some other effect and this has the result of improving the species overall fitness. A basic genetic algorithm then works as follows:

- Generate an initial population.
- Assess the fitness of each member of the population in relation to some criteria.
- Randomly select which members of the population breed, the probability of this happening being related to their fitness.
- Generate the offspring.
- Remove members of the population, the probability being inversely related to fitness, to keep the population at a given size.
- Repeat from stage 2 until some termination criteria is met.

The Population

In the case of an equation based modelling system, each member of the population is a possible solution vector. In the initial population each individual is randomly generated while in later generations they will be produced as a combination of their parents' data. In order to reduce the memory and population size requirements the variables are generally discretized and their range reduced as far as possible. Population size is usually held constant over time rather than being allowed to vary randomly. This

prevents population explosions resulting in huge memory and processor requirements or population reductions leading to a loss of variation and poor quality results. The population size is critical to the quality of the result likely to be obtained however it is to be remembered that this is a stochastic method and therefore results are never guaranteed.

Traditional GAs worked on a single population. Some research has investigated the use of multiple concurrent populations with limited transfer of individuals between populations. These are generally referred to as Island Hopping methods and attempt to reduce the tendency of populations to stagnate around a not particularly good solution. This is particularly common in relatively small populations or systems with a low mutation rate. Introduction of 'fresh blood' in the form of an individual or group from another population can provide the neccessary genetic variation to move the solution in a positive direction. It is not essential that the newcomers are fitter than the existing population per se, merely that they are different.

Fitness Criteria, Mating and Dying

In order to select which members of the population breed and die within a given generation, GAs make use of fitness functions. These are a measure of how suitable an individual member of the population is to the current environment. Fitness could for example be assessed as how close to zero, and therefore converged, the sum of the absolute values of the constraint violations is. All members in the population can mate and /or die in a given time-step. Often, the number of children produced in a given generation will be fixed and therefore an equal number of individuals will die. Exactly which individuals mate or die is determined using a roulette wheel approach, based on their fitness.

Generating Offspring

When the child is produced it inherits features (variable values) from both its parents. Most GAs will incorporate a small chance of mutation occurring, resulting in the child having data not held by either of its parents. Common methods of generating the child include either a random selection of values from the parent on a value by value basis or cross over of the values at a certain point. Some methods allow multiple crossover points or mixtures of the two methods.

Pure GA based approaches rely entirely on the random generation of children. It would be possible to adapt this step, given knowledge of the equation set, to only alter values that cannot currently be solved for using the available solution methods. Variables whose values can be determined would be fixed and therefore not affected by the GA.

6.6.2 EGG - Evolutionary Guess Generator

Initial development of a GA based initialisation method produced some promising results for models with few free variables. As the number of free variables increased however it became increasingly difficult to produce acceptable results. This is due primarily to the immense size of the solution space present for even relatively small numbers of variables at a coarse level of discretisation. It is suggested that future research focus on stochastic optimisation methods such as SQP.

6.7 Distillation Column Example

This section presents a potential initialisation method for a distillation column. The following steps are proposed:

- Perform an overall mass and energy balance;
- Derive internal profiles from user chosen method;

- Allow user analysis of results;
- Return values if acceptable.

Frequently, the initialisation process chosen will depend on what is known about the entity. In the case of a standard distillation column there are three material connections to the rest of the plant: the feed, the tops and the bottoms. Different approaches are required depending on which of these three streams are well specified. At least one stream should have been initialised previously but preferably two or more before the column can be initialised.

6.7.1 Overall Mass and Energy Balance

Ideally, two streams have been previously initialised when the column comes to be initialised. In this case the mass balance around the column is simple: either a summation of the outlet flows to give the feed or a subtraction of one outlet flow from the feed to give the other. This produces values which will be consistent with the surrounding units and should be close to the final solution.

Where only one stream is known, the user must supply sufficient extra information to perform the mass balance, supplying temporary specifications as in the sub-problem extraction method. One approach would be to:

- Assume sharp separation and identify heavy and light key components;
- Calculate resulting compositions and flows at top and bottom of the column;
- Determine bubble point temperatures at top and bottom;
- Calculate component enthalpy (h_i) and K_i values at top and bottom;
- Interpolate profile, e.g. linearly.

This gives values which are consistent with the surrounding units and a valid mass balance for the column but will probably be at best a poor approximation to the final

result. This option is particularly sensitive to the user's knowledge of the process and is in effect a hot start for the column from a different process. The worst case is where all three streams have been initialised. The mass balance therefore becomes a check on the surrounding initialisations. In the unlikely event that the flows balance this is not a problem and the process continues. Where there is a mismatch the user must decide which stream or streams to alter in order to balance the column. Inevitably, this results in a break between the column and up or downstream units but at least produces a valid mass balance for the column.

The overall energy balance can be performed at the same time.

6.7.2 Column Internals

The user would then be presented with a number of options for completing the rest of the initialisation. In the case of the rigorous, tray by tray column model this can involve a substantial amount of calculation. Possible options include:

- Simple. All compositions and flows as feed;
- Linear. Linear interpolation between feed stream and tops or bottoms as applicable;
- Lewis-Matheson (Lewis and Matheson, 1932). Standard L-M method, applied from tops to feed and then bottoms to feed;
- Scaled Lewis-Matheson. As above but calculation stops at a tray where the liquid composition (probably defined as light key/heavy key ratio) is similar to the feed. The profile from the L-M method is then scaled to fit the actual number of trays in the column.

6.7.3 Analysis of Results

Another advantage of having the initialisation routine connected to a specific EType is that the application can provide an analysis of the results. In the case of the column, the user may have supplied temporary specifications which result in trays running dry or which on viewing the results do not fit with the rest of the process. Because the application is column specific, it can be written to identify problems particular to columns (such as trays running dry) and can suggest alternative values for the temporary specifications. The user is free at this point to change any values they wish and to re-run the initialisation using any of the methods provided.

Once they are satisfied with the results the values are returned to FMS and the initialisation process continues with the remaining units.

6.8 Summary

The core routines and structures required to implement FMS_INIT are present within JFMS and models have been written incorporating user written initialisation methods. Further work is required to validate the proposed approach. This should be considered a necessary piece of research as current methods are time consuming and frequently ineffective.

Chapter 7

Discussion and Future Work

7.1 Evaluation of FMS

7.1.1 Overview

FMS provides a object-orientated modelling environment for rapid development and evaluation of modelling tools and processing techniques. It has been used extensively within the department for this purpose at both undergraduate and postgraduate level. In effect, it acts as an interface between applications, the core model data and the user written methods embedded in the model.

FMS has adapted to incorporate the feedback from these other projects and has therefore developed beyond the initial, hard-coded utility system modelling package. The environment as it stands it truly generic and readily extensible both at a model definition level and in terms of functionality as provided by attached applications and internal methods. It satisfies the requirements for modelling environments as described in chapter 2, providing the ability to:

- Develop and build large and complex models;
- Access and manipulate model data;
- Apply methods and applications to the model;

- Add new methods and applications;
- Add new data structures;
- Operate as a stand-alone application or provide modelling capability within an existing system.

7.1.2 Use of FMS

Throughout the development of FMS it has been used by other researchers within the School of Chemical Engineering to satisfy their modelling or research needs. The most notable of these is a recently completed PhD thesis (Rodriguez-Toral, 1999). A number of final year honours research projects have also been carried out. These have evaluated FMS as a modelling language both in terms of functionality and usability and produced some novel applications and extensions to the language as well. An MSc project (Felton, 1996) used an early version of FMS to test models before developing hardcoded Fortran90 routines to continue the work. This section outlines the work performed in these projects and the effect they have had on the development of the system.

Synthesis and Optimisation of Utility Systems

As this project's focus moved more towards the actual modelling language and environment rather than the modelling of utility systems a colleague took over this area of research. This project focussed on the modelling of combined heat and power systems, an area of increasing industrial interest as efficiency gains through heat integration become more and more important. Typically these systems involve complex networks of heat exchangers, compressors, turbines and associated units and so they are ideally suited to the equation based modelling approach.

Two main areas were studied: the optimisation of fixed structures with in excess of 3000 variables using Sequential Quadratic Programming (SQP) methods (Rodriguez-Toral et al., 1999a) and the synthesis of similar sized problems using an MINLP (Mixed

Integer Non-Linear Programming with Branch and Bound) solver (Rodriguez-Toral et al., 1999b). During the course of the project a large number of models were developed but the main area of interest as far as this report is concerned was the addition of integer variables to FMS to allow an MINLP solver to be connected by an independent user. This highlights the extensibility of the language and shows that the initial requirements of the project were met, namely that:

- The system should allow easy addition of new applications.
- It should be possible for the user to extend the functionality of the language, either through adding new keywords to the input language or allowing additional modelling information to be provided at the Application Server level.

Current versions of FMS have the facility to handle integer variables. Use of data files or other input at the application server level is a useful method of satisfying highly application specific modelling requirements or to rapidly construct and test new data structures. This avoids having to alter the JFMS core, leaving the basic modelling and model manipulation routines untouched. It also allows users unfamiliar with Java or C programming to extend the language without having to either extend existing code or indeed produce new code in an unfamiliar language.

Honours Research Projects

The first FMS undergraduate project (Griffiths, 1997) concluded that FMS provided a useful modelling tool but required extensive changes to the user interface. This led to the development of the GUI used in JFMS and all subsequent projects have been based on this variant of the system.

A problem that occurred frequently during this project and indeed in subsequent work was errors being made in the equation subroutines. Generally this would be incorrect calculations for first or second derivatives, either in the actual value of the derivative or in its position in the vector. Occasionally this was the result of incorrect differentiation

in the first place although more commonly it was simply a coding error. Traditionally, the data would have to be analysed manually and in a large model this is extremely time consuming.

Two projects have looked at this problem (Farrell, 1998) and (Iordanis, 1999), producing an equation routine verification package and an automated equation routine writer respectively. Work by Farrell produced an equation routine verification package called MICE (Model Information Checking Engine). This package verified the supplied first and second derivative information by performing finite difference style perturbations to the variable set and comparing the result to that predicted by the residual equation. A stand alone equation routine generator was produced in Maple by Iordanidis. This uses Maple to find derivative information for a function f(x) [in f(x) = 0] and produces the Fortran90 code for the methods required by JFMS. Some changes are needed within JFMS to fully implement the code produced. These applications are useful tools and have the potential to dramatically reduce the time required to produce working models.

Other recent projects have evaluated the proposed initialisation methodology (Cochrane, 1999) and used JFMS to evaluate optimisation methods (Verbeek, 1999). FMS provides a useful tool, both to develop complex models of processes and to develop and test new applications. These projects have assisted in the development process and have proved that the initial aims of the work have been satisfied.

Applicability of FMS

The approach described in this thesis is appropriate for users familiar with the concepts of equation based modelling and with a reasonable level of mathematical programming ability. FMS has been successfully used in advanced projects at both PhD and MSc levels. These projects have required extensive user involvement, including:

- Complex model development;
- Development and addition of new methods and applications;

- Addition of new language constructs.

Therefore, it has satisfied the primary requirement for the work: to support the development of novel processing, solution and optimisation methods by researchers with post-graduate or more advanced experience in the field.

While the system is usable by undergraduates, addition of new methods and applications has proven difficult. This is due mainly to the levels of abstraction required within the model definition and storage in order to guarantee true flexibility. This is further complicated by a lack of familiarity with scientific programming in general and, specifically, equation based modelling. However, feedback from undergraduate users has indicated that FMS can be used effectively to develop complex models with relatively little support.

7.1.3 Use of Methods

Methods have proven an extremely powerful and flexible way of embedding functionality within models, supporting the predictions made within the ASCEND project (Abbott, 1996). Providing a common interface to represent equations of any complexity, access external applications and databases or call task specific user written subroutines from a generic model is both useful and novel. The ability to reuse existing, non-JFMS format models widens the scope for this approach immensely and answers the requirement identified by Westerberg (Westerberg and Benjamin, 1985) to reuse existing models, regardless of format. Given even a limited knowledge of Fortran or C it is possible for the user to customise the behaviour of their models within a standard language and gain access to existing models and applications.

Once written, methods are easily re-used, providing a known and debugged capability to other, possibly less experienced, users. This must be balanced against the increased overhead in terms of development of specific methods when compared to the current standard packages supporting direct entry of equations assisted by equation parsers and automatic differentiation.

While most equation-based modelling environments such as ASCEND and gPROMS started out with relatively simple languages and a 'no-coding' philosophy, as their functionality increases so does their similarity to a high level programming language. The enhanced modelling support derived from such object-orientated, hierarchical approaches cannot be denied but it can also be argued that the aim of producing an 'out of the box, no coding required' modelling language has produced a high level programming language of its own, albeit one tailored to the description of equation-based models. It is interesting to see interfaces for such packages being incorporated into process design environments as this is typically intended to reduce the complexity of the modelling process from the user's perspective.

Analysis of large equation sets produced by such hierarchical modelling languages (Allan and Westerberg, 1999) suggests that for a system of 100,000 equations typically less than 100 unique equation forms will exist. The method proposed searches the model to determine the minimal set of equations required to represent the process and then compiles these into C code for use in ASCEND IV. This relates well to our own experience that suggested that after the initial work in writing methods is complete only very application specific methods were developed.

In order to support the reuse of methods and models it is essential that users have access to properly documented libraries of existing code. This was done in an informal way within the department but it is clear that this aspect of modelling needs to be formalised (Allan, 1998).

7.2 Future Work

7.2.1 Modelling Environment

Thin Client Architectures

Modelling environments should take account of the developments being made in network computing and Internet based applications in general. FMS provides an initial

basis for developing a true thin client based modelling capability.

The rigid separation of the GUI and model definition modules from the processor intensive underlying analysis and solution methods allows use of cheap, low powered PCs to perform the user intensive model definition and manipulation. The processor intensive work can be performed on a remote , high powered computer with the required applications and licenses. Such an approach allows many users to share the costs of expensive items such a high-performance computers and licenses for advanced software applications whilst maintaining the ability to work in a distributed fashion.

Equation Representation

All equations within FMS are represented using methods. While this approach has been proven to be extremely effective at embedding complex functionality within models, it is arguably inappropriate for describing simple equations.

Most existing modelling environments provide the user with the ability to define an equation directly within a model, supporting this functionality with underlying parsing and automatic differentiation routines. This reduces the time spent in initially defining the equation but looses the efficiencies gained in terms of code re-use and debugging. A hybrid approach is therefore recommended, allowing the user to define simple equations within the modelling language while retaining the ability to embed methods in order to represent more complex functionality.

7.2.2 Initialisation Methods

The initialisation methods outlined in chapter 6 are promising but, due to lack of time have not been properly investigated. Use of an object orientated modelling language such as FMS, ASCEND or gPROMS provides a structure to the resulting variable and equation sets that can be exploited. Decomposition of the model into its' component blocks allows solution or initialisation on a block by block basis, mimicking the processes currently used by modellers to derive initial values for their models. Automation of

this process is a logical starting point for computer based initialisation methods.

Chapter 8

Conclusions

The approach adopted is suitable for users with a computing and equation based modelling background and supports the development and evaluation of complex and novel modelling tools and model types. While it can be used by less experienced users to develop and manipulate existing models it is not really designed to be used in this style. FMS is best used to experiment with different model formulations or to rapidly develop, connect and test novel processing techniques requiring new data structures or language features not available in the more main stream modelling environments.

This extensibility arises from the design of the application. FMS as the core, unextended environment provides a means to represent only high level information about a model; the components in the model, the public variables and specifications, the ability to declare methods which interact with the model in some way and an interface to link external applications to. This is effectively the minimum set of data required to specify and manipulate a process model, with FMS acting as the interface between the model data, methods and applications.

As has been demonstrated by other languages in the field, most real advances in model formulation, solution or optimisation methods require at least some modification of the modelling language. This is frequently time consuming, complex and best done by the original development team. In a situation where the changes to be made are rapidly changing as research develops such an approach is not realistic. It is envisaged that an

environment such as FMS be used to evaluate the changes during development with the main stream modelling environment only being updated once the changes are finalised. Research effort should focus on research, rather than on extending the tools.

The environment and language's functionality is determined by the available methods and attached applications. The functionality provided by the methods is that of a full featured, high level programming language such as FORTRAN90 or C rather than the subset (simple do loops and possibly conditional blocks) that are typically mapped into a modelling language. The FMS language itself is therefore extremely simple and stable but a user does require the ability to program in a high level lnaguage.

One of the key requirements identified by researchers in the field is the ability to reuse models. Given the number of existing applications, models and databanks any closed modelling environment that attempts to enforce a single model format will not be used. The expense of converting and debugging large models is simply too great and therefore the ability to handle mixed format models is critical.

Methods provide a powerful way of embedding complex functionality within a simple language. This includes:

- Analysis routines;
- Initialisation routines;
- Calls to external applications and libraries;
- Description of equations;
- Model specific output;
- Calls to non-standard format models;
- Internal solution steps.

It is hoped that FMS will prove useful within the department and continue to be used to support the development of novel solution and processing tools. This thesis presents

the development of FMS, its data structures, modelling language and method and application interfaces with the intention that it be usable by future researchers.

Appendix A

JFMS User Guide

A.1 Introduction

This appendix contains the user guide for JFMS, the Java based Flexible Modelling System. Model building is discussed in chapter 5, further appendices detail method writing and the addition of new applications.

JFMS can be used as both a stand-alone modelling package and to provide modelling capability within another application. The following sections outline the steps required to use JFMS in both modes.

A.2 Software and Operating System Requirements

JFMS runs under Solaris 2.5.1. It requires the following software:

- Java 1.1 or newer;
- Fortran90 compiler (currently epcf90);
- C / C++ compiler (currently gcc);
- XEmacs.

A.3 Installing the Software

The current version of JFMS is stored in /home/dave/jfms_arch.

In order to assist with the installation process two batch files have been created. These are *importFMS* and *compFMS* and are stored in /home/dave/FMS_bits. The user should create a FMS directory within their own account and run first *importFMS* and then *compFMS* in their FMS directory. This will create the necessary directory structure for the basic installation of JFMS, copy across the relevant files and compile the package locally.

The basic installation provides:

- GUI;
- Model building routines;
- Application server;
- NLAE Solver;
- Equation Analyser;
- Filter SQP;
- NAG Sparse Linear Solver.

A.4 JFMS as a Stand-alone Modelling Package

A.4.1 Starting the Application

Start JFMS by typing **java FMSTop** in the local FMS/GUI directory. It is recommended that this be done through an XEmacs shell as this allows easier control of any text output from the package or attached applications.

The Control Panel, as illustrated in figure A.1 is the window that appears. Initially all the lists (Variable Types, Entity Types and Models) will be empty apart from the **[Add New]** entries.

A.4.2 VTypes and ETypes

Adding to the GOL - Generic Object Library

VTypes should be added before ETypes as some initial parsing is done when ETypes are entered into the system.

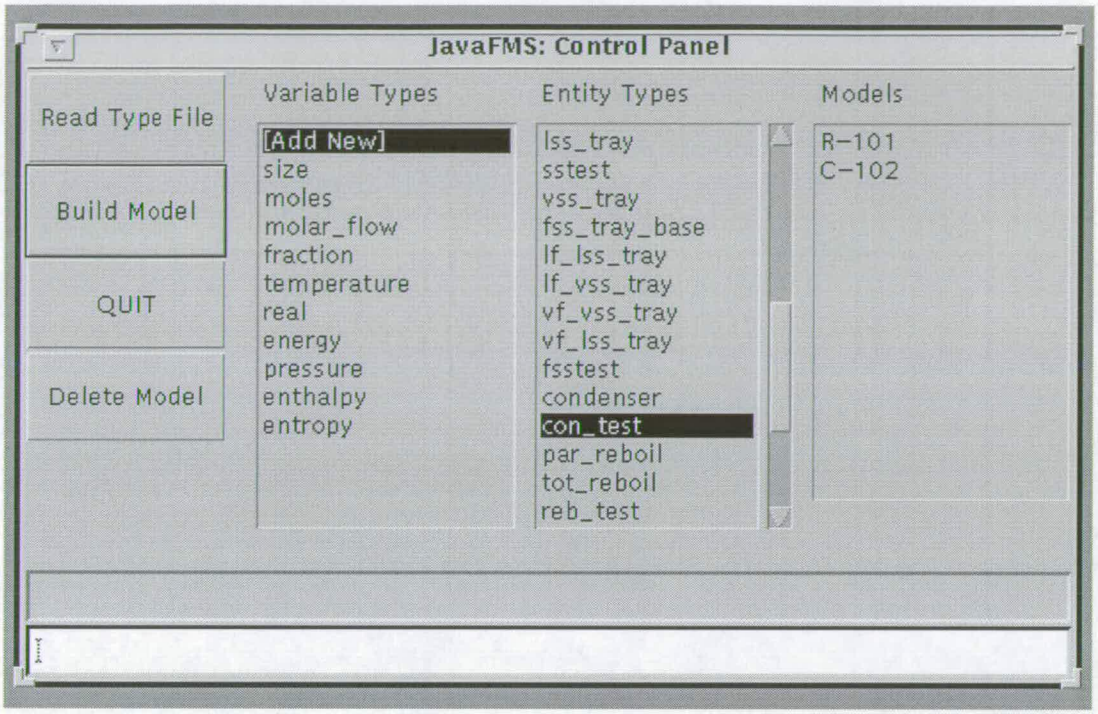

Figure A.1: JFMS Control Panel

Typically, VTypes and ETypes will be defined in existing text files (.vt and .et respectively). These can be added into the system through the File Handler window, opened by clicking on the **Read Type File** button on the Control Panel. File names can be entered directly into the text box or can be located using the **Browse** button. In the browser, the left column shows available .vt and .et files in the current directory while the right one shows the currently selected files. The user can navigate through the directory structure and select multiple .vt and .et files. Files are chosen by selecting the filename in the left hand list and clicking on the **Add Selected** button. When the selection is complete the user should click the **OK** buttons until focus returns to the Control Panel.

VTypes and ETypes can also be added to the library by double-clicking on the [**Add New**] entry in the relevant list. This initiates the VType Editor (figure A.3) or EType Editor (figure A.4) as appropriate.

Editing and Viewing VTypes and ETypes

Double-clicking on an existing VType or EType initiates the appropriate Editor. These allow the user to view and edit existing types and create new ones. It is important that new types are given unique names. New types defined are saved in a .vt or .et file of the same name for later re-use.

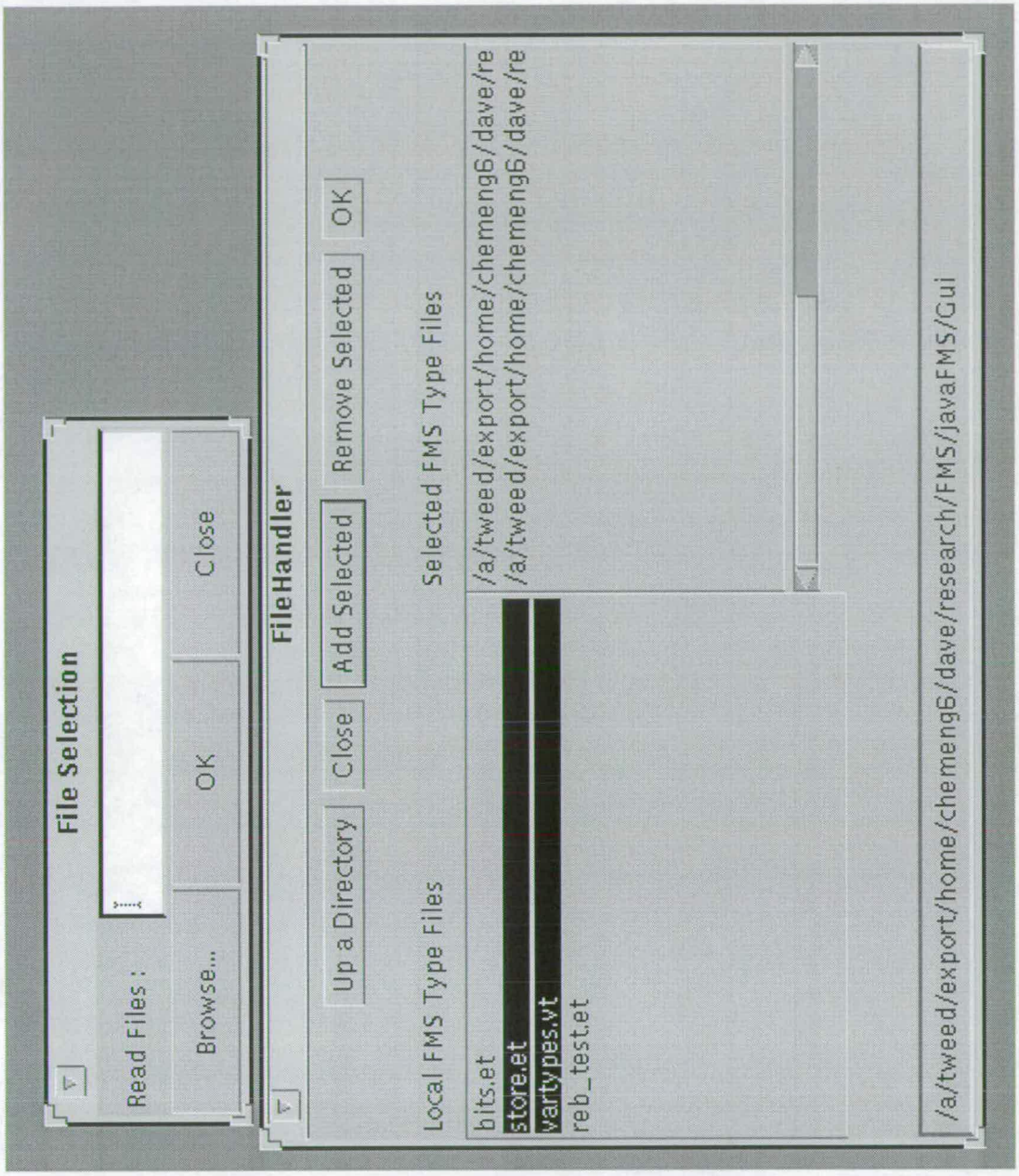

Figure A.2: File Handler

A.4.3 Dealing with Models

Building Models

Model ETypes (see page 49) can be compiled into a usable model by selecting the required EType and then clicking the **Build Model** button in the Control Panel. JFMS asks the user for a unique model name, compiles the EType and the model

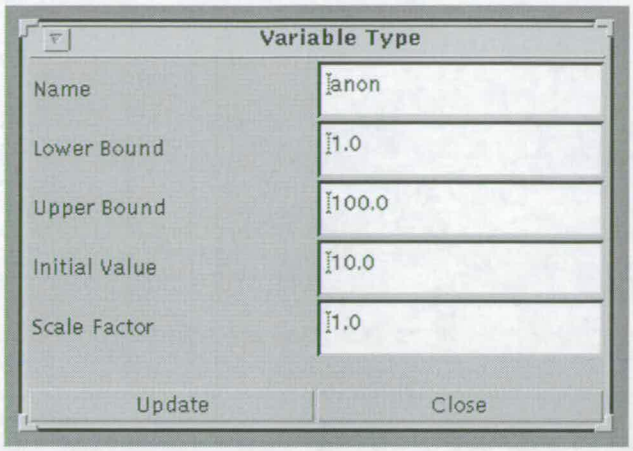

Figure A.3: VType Editor

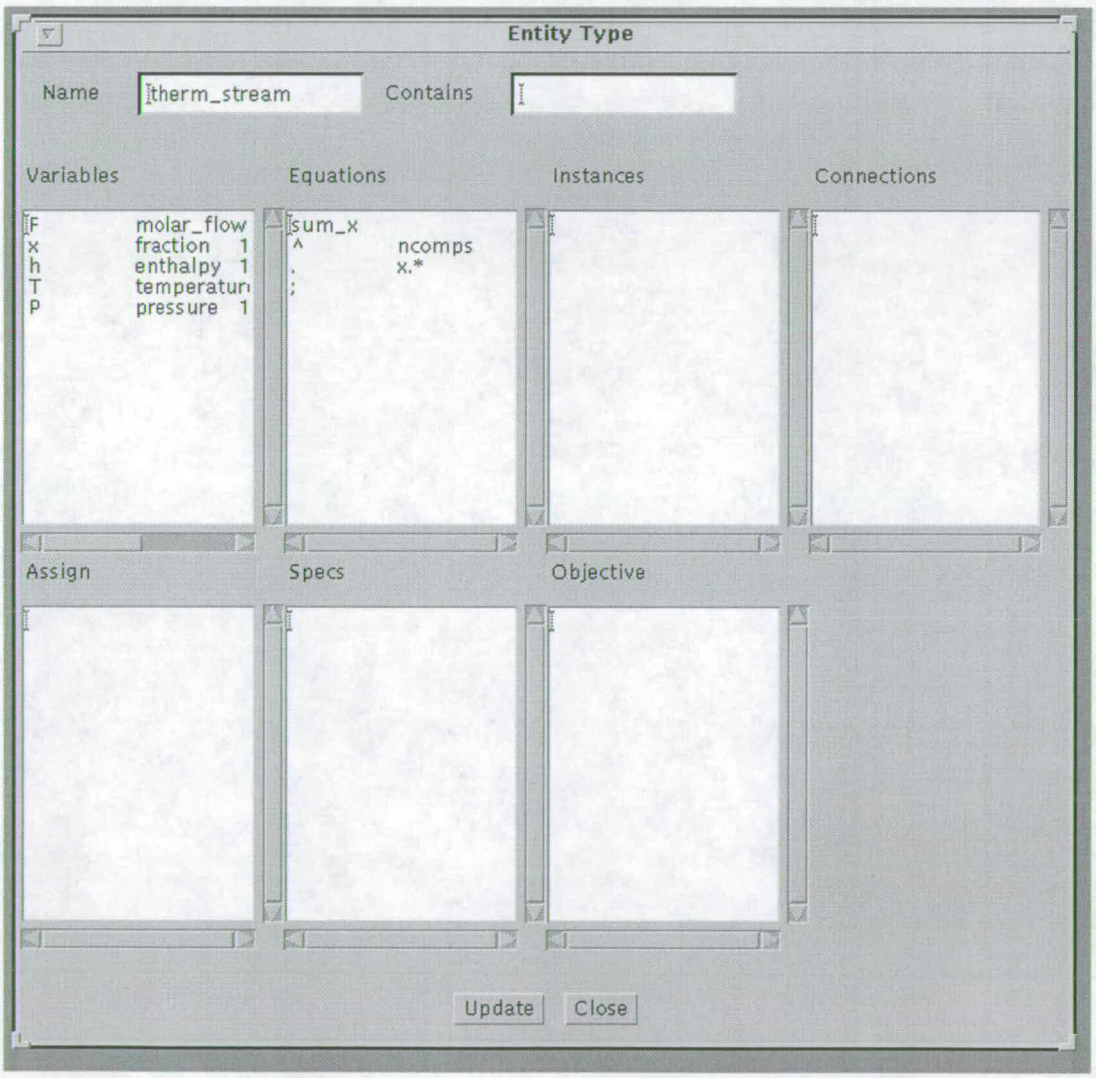

Figure A.4: EType Editor

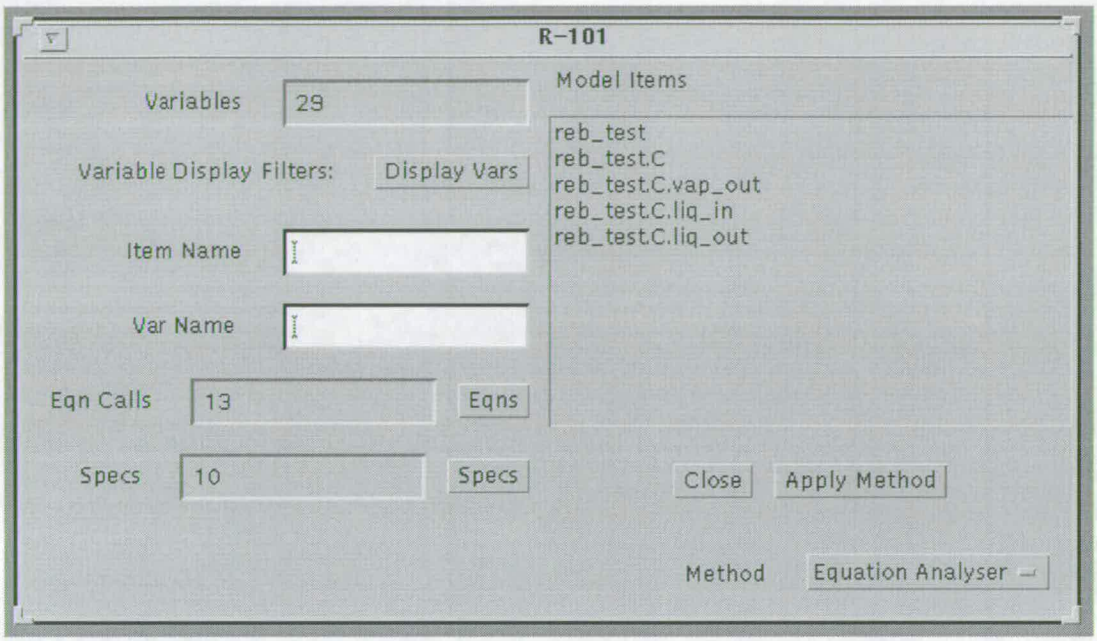

Figure A.5: Model Handler

appears in the Models list. Multiple models can be held in the NGOL vector allowing the user to switch between models quickly. Double-clicking on a model name in the Models List opens the relevant Model Handler (figure A.5).

Manipulating Models

The Model Handler shows:

- Number of variables;
- Item and Variable Name filters to apply to the variable set;
- Number of Equation (method) calls;
- Number of Specifications;
- Item List;
- Pull down list of available applications.

The Item List is a list of the component models within the global model. Double-clicking on an item opens up the Variable Editor (figure A.6) and displays the variables for that component. This has the same effect as entering the item name in the Item Name text box and then clicking on the **Display Vars** button. Item and Var Names

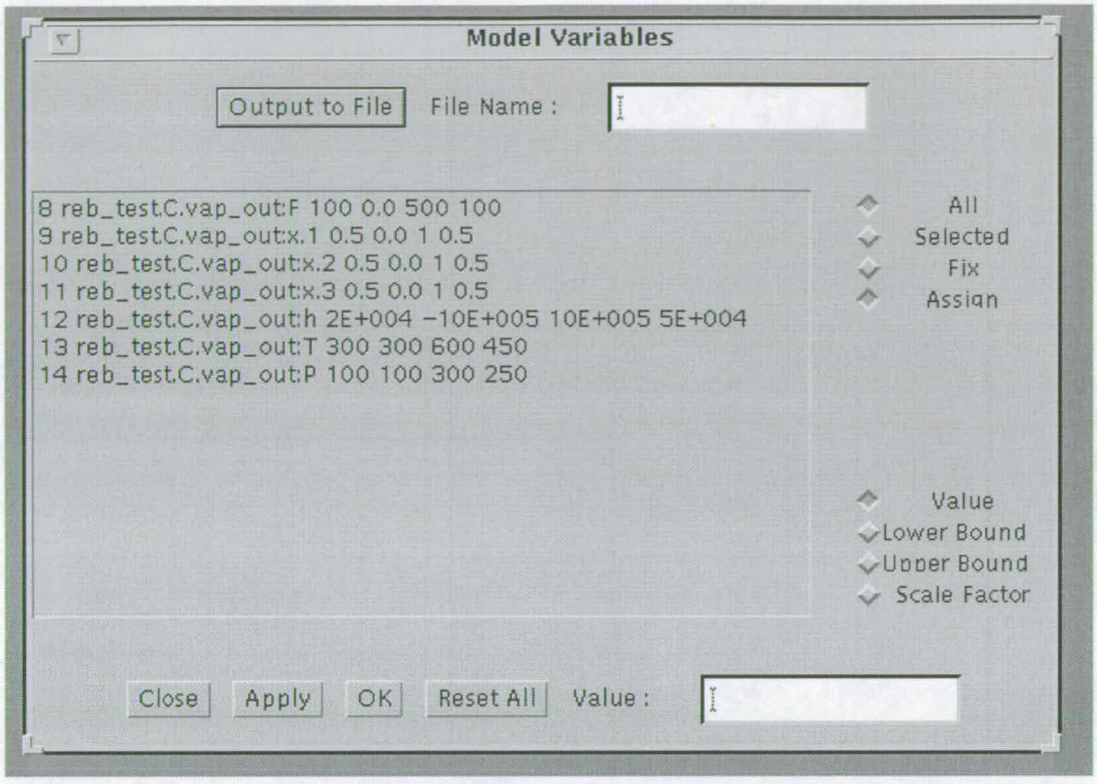

Figure A.6: Model Variable Editor

can be specific instances such as reb_test.C and reb_test.C.liq_in or wild-card based such as *.liq_in to find all components whose name ends in liq_in.

The Variable Editor allows the user to view and alter the variable set or to output the current view to a file. Variables are chosen from the list on the left and buttons on the right control what happens when the changes are applied. Changes can be made to either all the variables or just those currently selected, to value, upper and lower bound or scale factors. Specifications can also be declared through this editor or the Specifications Editor (figure A.7). The display shows: variable id number, item name, variable name, current value, lower bound, upper bound and scale factor for each variable.

Calling Applications

The current model can be passed to an application such as a solver or optimiser. The application is selected from the pull down list and then initiated by pressing the **Apply Method** button. The model data is then transferred to the Application Server and control passes to the applications own interface. When processing finishes, the focus returns to the Model Handler window.

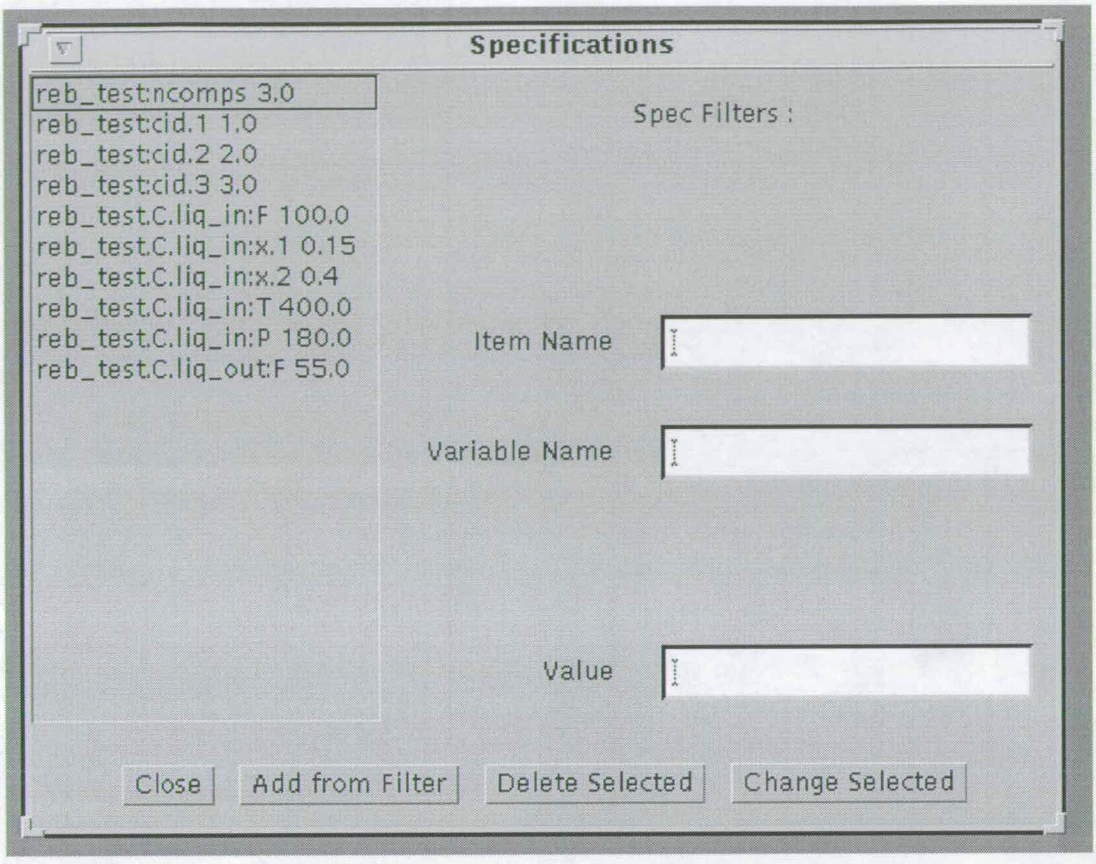

Figure A.7: Model Specifications Editor

A.5 JFMS as an embedded Modelling Capability

JFMS model building and processing routines can be called from within a host application. The host application must have access to the local FMS directories and will call java methods from within the GUI directory. The embedded modelling capability can be provided in two forms:

- Using the JFMS Model Building Routines and Application Server;
- Using the Application Server alone.

A.5.1 Using the JFMS Model Building Routines and Application Server

Where the user wants to produce models using the JFMS MDL it is necessary to use both the JFMS Model Building routines and the Application Server. It is assumed that the .vt and .et files are already written or are produced by the host application.

```
toyBox = new ModelBox();
toyBox.g0.readVTF ( ``{\em vtypes.vt}'' );
toyBox.g0.readETF ( ``{\em etypes.et}'' );
toyBox.createNGOL ( ``{\em R-101}'' , "{\em Reboiler}" );
toyBox.execProcess ( ``{\em R-101}'' , ``{\em process}'' , \newline ``{\em results.
```

Figure A.8: Code Required to Execute JFMS Models from a Host Application

The functions required to initialise and build a model are held within FMS/GUI. The function calls in figure A.8:

- Initialise the model library as a structure called *toyBox*;
- Read VTypes from a file *vtypes.vt*;
- Read ETypes from a file *etypes.et*;
- Create a model called *R-101* from the *Reboiler* EType;
- Pass the model to the Application Server for processing by application *process*;
- Output the results to a file *results.dat*.

A.5.2 Running the Application Server Alone

The data structures in the Application server are a mirror of those in the core model definition described in chapter 4. If so desired, these can be directly populated by the user, avoiding the need to produce JFMS .vt and .et files or use the provided model building routines. This allows the user access to all the applications and methods and allows JFMS to be used as a modelling capability within another modelling package. In this role, the user obtains Jacobian, hessian nad residual values from the JFMS model which are then integrated with the host modelling environment for solution.

Appendix B

Methods

B.1 Method Library

The user written methods are accessed through a Fortran90 subroutine called eqn_handler. This routine is held in the FMS/F90FMS local directory and is based around a case construct. Methods were originally used solely to represent equations, hence the subroutine name. Eqn_handler takes the form shown in figure B.1, new methods should be added in the same format as the equality and sum_x methods shown in the figure.

Where a large number of methods have been declared re-compilation becomes time consuming and, depending on the compiler used, the CASE structure may not be able to handle the number of options. This can be avoided by the use of prefixes identifying families of methods and altering the Method library as shown in figure B.2. The subroutines *std_methods* and *ml2_methods* are themselves method libraries.

```
subroutine eqn_handler ( jpt , eq_name , iv , rv , nj , nh , &
     r , rc , jc , hc , oc )
 use add2jhl
 type ( t_list ) :: jpt
 integer   :: iv, nj , nh , r
 real    :: rv
 character(100)  :: eq_name
 logical   :: rc , jc , hc , oc

 select case ( eq_name )

 case ( 'equality' )
  call equality ( jpt , iv , rv , nj , nh , r , rc , jc , hc  , oc )

 case ( 'sum_x' )
  call sum_x ( jpt , iv , rv , nj , nh , r , rc , jc , hc  , oc)

 case ( 'none' )
  !! ignore - no eqns declared

 case default
  print*,'No such method : ', eq_name
  stop

 end select

end subroutine
```

Variable Name	Description
jpt	Internal JFMS variable
eq_name	Method name
iv	Pointer to the variable list for the method within the fms_eqn vector
rv	Real value
nj	Current number of Jacobian elements
nh	Current number of Hessian elements
r	Current number of equations declared
rc	Logical variable controlling whether residual is calculated
jc	Logical variable controlling whether Jacobian is calculated
hc	Logical variable controlling whether Hessian is calculated
oc	Logical variable controlling whether function value is calculated

Figure B.1: Basic Method Library Format

B.2 Writing Methods

Methods are used to embed functionality within the ETypes and resulting component models. There are two classes of method, namely:

```
select case ( eq_name(1:3) )

case ( 'STD' )
 call std_methods ( jpt , eq_name , iv , rv , nj , &
nh , r , rc , jc , hc , oc )

case ( 'ML2' )
  call  ML2_methods ( jpt , eq_name , iv , rv , nj , &
nh , r , rc , jc , hc , oc )

case default
 print*,'No such method / family : ', eq_name
 stop
end select
```

Figure B.2: Large Method Library Format

- Methods Representing Equations;
- Methods as an Interface to External Packages and Models.

B.2.1 Methods Representing Equations

Many methods are used to define equations. This section provides a standard template for such methods and gives examples of equation methods. There are three core JFMS functions to handle Jacobian, Hessian and residual data. Each function adds a new element to the appropriate vector for the global model. Use of these functions is demonstrated in the following examples but for reference the calling statements are:

- rhs_list (jpt , r , val , sv)
- jac_lists (jpt , nj , r , col , val , elt)
- hess_lists (jpt , nh , row , col , r , val)

The logical flags jc, rc, hc and oc were initially intended to optimise the processing of large models in applications that did not always require all of the data to be recalculated. While this is still usually the application of these flags, it is possible to use them simply as control flags in methods not dealing with Jacobian and similar data.

Equation Method Template

```
subroutine template (jpt , iv , rv , nj , nh , r , rc , jc , hc  , oc)
```

```
use add2jhl
type ( t_list )  :: jpt
integer   :: iv, nj , nh , r , elt , row
real    :: rv
character   :: eqt
logical   :: rc , jc , hc , oc

!! Increment number of equations

r = r + 1

if ( rc ) then

 !! Calculate residual if rc = true

 val = [calculate residual]
 sv = [calculate eqn scale factor]
 call rhs_list (  jpt , r ,  val , sv)

else
end if

if ( jc ) then

!! Calculate Jacobian if jc = true

!! Increment number of Jacobian elements

 nj = nj + 1

 col = [set variable number]
 val = [calculate derivative]
 elt = [set to 1 if fixed, 0 if variable]
 call  jac_lists ( jpt , nj , r , col , val , elt )

else
end if

if ( hc ) then

!! Calculate Hessian if hc = true

!! Increment number of Hessian elements

 nh = nh + 1
 row = [set row number]
 col = [set column number]
 val = [calculate second derivative]
```

```
 call hess_lists ( jpt , nh , row , col , r , val )

else
end if

if ( oc ) then

!! Return value of x

 rv = [calculate function value]

else
end if
end subroutine
```

Equation Method: a - (b/c) = 0

```
subroutine amb_c (jpt , iv , rv , nj , nh , r , rc , jc , hc  , oc)
 use add2jhl
 type ( t_list )  :: jpt
 integer   :: iv, nj , nh , r , elt , row
 real    :: rv
 character   :: eqt
 logical   :: rc , jc , hc , oc

!! Declare extra variables as necessary

 real :: a , b , c , val , sv , as , bs , cs
 integer :: col , apos , bpos , cpos

 sv = 0.0

!! Determine which variables are to be processed.
!! Call for this routine within the EType takes the form:

!! EQNS
!! amb_c
!!  . a
!!  . b
!!  . c
!! ;

!! These entries are parsed and the specific variable id number stored
!! in fms_eqn. When a method is called, iv stores the index before the
!! first of the relevant entries in fms_eqn. The first variable id number
!! is therefore stored in fms_eqn ( iv + 1) and so on.
```

```
!! apos, bpos and cpos are declared and their values calculated for
!! convenience. These are known as VARIABLE INDICES.

 apos = fms_eqn ( iv + 1 )
 bpos = fms_eqn ( iv + 2 )
 cpos = fms_eqn ( iv + 3 )

!! Each variable has a value and local variables a, b and c are declared to
!! store the current values from the global variable set, fms_vars. This is
!! referred to as UNPACKING

 a = fms_vars ( apos ) % value
 b = fms_vars ( bpos ) % value
 c = fms_vars ( cpos ) % value

!! If the fms_sys%recalc flag is true then the equation set is being rebuilt.
!! Scale factors for the variables and the equation are determined the first
!! time round.

 if ( fms_sys%recalc ) then

  as = fms_vars ( apos ) % scale
  bs = fms_vars ( bpos ) % scale
  cs = fms_vars ( cpos ) % scale

  if ( c /= 0.0 ) then
    sv = max ( abs(fms_vars(apos)%scale) , &
       abs ( fms_vars(bpos)%scale / fms_vars(cpos)%scale ) )
  else
   sv = abs(fms_vars(apos)%scale)
  end if

 else
 end if

!! Before an equation is added, the row counter r should be
!! incremented. This applies when adding Jacobian and Hessian]
!! elements as well (nj and nh).

 r = r + 1

!! If jc is true then calculate the Jacobian elements

 if ( jc ) then

!! This is the derivative of the function wrt variable a
!! nj is the Jacobian element number;
```

```
!! col is the column in the Jacobian that the element belongs in;
!! val is the derivative value;
!! elt is set to 1 as the derivative is a fixed value
!!         (set to 0 if it is variable)

!! This is repeated for each variable in the equation.

  nj = nj + 1
  col = apos
  val = 1.0
  elt = 1

!! Each element needs to be added to the global Jacobian vector.
!! This is done by calling jac_lists(). Similar functions exist for
!! residual and Hessian elements.

  call  jac_lists ( jpt , nj , r , col , val , elt )

  nj = nj + 1
  col = bpos
  val = -1.0/c
  elt = 2
  call  jac_lists ( jpt , nj , r , col , val , elt )

  nj = nj + 1
  col = cpos
  val = b / ( c*c )
  elt = 2
  call  jac_lists ( jpt , nj , r , col , val , elt )

 else
 end if

 if ( rc ) then

!! If rc is true then calculate the residual

  val = a - b / c
  call rhs_list (  jpt , r ,  val , sv)

 else
 end if

 if ( hc ) then

!! If hc is true then calculate Hessian contributions

  nh = nh + 1
```

```
  row = bpos
  col = cpos
  val = 1.0 / ( c * c )
  call hess_lists ( jpt , nh , row , col , r , val )

 else
 end if
end subroutine
```

Equation Method: Equations involving vectors

This example illustrates the recommended approach to dealing with equations involving vectors. The code has been simplified to concentrate on the vector handling.

```
!! The equation is: F - sum ( molar flows ) = 0 and is declared as:
!! EQNS
!!  mfsum
!!   ^ ncomps
!!   . F
!!   . mf.*
!!  ;

!! fms_eqn therefore holds indices for :
!! ncomps, F, mf.1, mf.2, ..., mf.ncomps

!! declare variables as before.

integer :: ncomps, mf_base , F_base, comp , col
real :: mf , F , sum , val, sv

!! ncomps does not appear in Jacobian etc and so only its value
!! need be stored.

ncomps = nint(fms_vars(fms_eqn(iv+1))%value)

!! F_base is the variable id number for the flowrate variable.

F_base = fms_eqn(iv+2)

!! mf_base is set to the index before the 1st molar flow. The index for
!! mf.1 is therefore mf_base + 1 and so on.

mf_base = iv + 2

!! Determine current value of F.
```

```
F = fms_vars(F_base)%value

sv = 0.0

r = r + 1

if ( rc ) then

 !! Calculate residual

 sum = 0

 do comp = 1 , ncomps

!! A value for a specific molar flow is determined using the
!! format below:

  mf = fms_vars( fms_eqn ( mf_base + comp ) )%value

  sum = sum + mf

  sv = max ( sv , abs (fms_vars( fms_eqn ( mf_base + comp ) )%scale ) )

 end do

 val = F - sum
 sv = max ( abs ( fms_vars(F_base)%scale ) , sv )
 call rhs_list (  jpt , r ,  val , sv )

else
end if

if ( jc ) then

 !! Calculate Jacobian

 nj = nj + 1
 col = F_base
 val = 1.0
 elt = 1
 call  jac_lists ( jpt , nj , r , col , val , elt )

 do comp = 1 , ncomps

  nj = nj + 1
  col = fms_eqn ( mf_base + comp )
  val = -1.0
```

```
  elt = 1
  call  jac_lists ( jpt , nj , r , col , val , elt )

 end do
 else
 end if
end subroutine
```

Equation Methods Summary

JFMS provides the three core functions required to create Jacobian, Hessian and residual vectors for the global model. A method is called from a specific component in the global model with a pointer in to the fms_eqn vector which provides the indices for the specific variables. These indices can be used to locate the specific variables within the variable set, fms_vars.

The structure used within the method subroutine is user defined, the previous examples demonstrate structure that has been used within the group and should therefore be considered as illustrative rather than prescriptive. Equation methods can therefore be written to suit the application that they are intended for - if Jacobian, Hessian or residual vectors are not required then there is no need to code them.

B.2.2 Methods as an Interface to External Packages and Models

Interface methods are called from components in the global model in exactly the same format as equation methods. In this instance however, processing is to be done by an external package or model rather than from within JFMS. This process is described at a high level below and then illustrated using an example method linking to an external physical property package.

As with equation methods, the exact form and purpose of the interface method must be tailored to suit the users requirements. Assuming that the external package is being used to return Jacobian, Hessian and residual data the steps required are:

- Declare any temporary variables required;
- Derive variable indices and unpack the variables as described in example 1;
- Convert variables to the appropriate units used in the external package;
- Translate variables from JFMS variable format to the format required in the external package;
- Call the external package;
- Translate results back to JFMS variable format;
- Convert variables back to the units used by JFMS;
- Store Jacobian, Hessian and residual values as before;
- Delete any temporary variables.

Calculation of Liquid Enthalpy Using an External Method

```
subroutine liquid_enth ( jpt , iv , rv , nj , nh , &
   r , rc , jc , hc  , oc)
 use add2jhl
 type ( t_list )  :: jpt
 integer   :: iv, nj , nh , r , elt , row
 real    :: rv
 character   :: eqt
 logical   :: rc , jc , hc , oc

!! Declare temporary variables

 real , allocatable :: dhcalcdx(:)
 real , allocatable:: x(:)
 real :: hcalc, dhcalcdt, dhcalcdP, hl,T ,P ,real_val, val , sv
 integer :: ncomps, comp , col , hlpos, Tpos
 integer :: Ppos, xpos , xbase , idbase
 integer , allocatable :: compindex(:)
 character :: phase

 sv = 0.0

 ncomps = nint(fms_vars(fms_eqn(iv+1))%value)
```

```
 allocate ( compindex ( ncomps ) )
 compindex = 0.0

 allocate ( x ( ncomps ) )
 x = 0.0

 allocate ( dhcalcdx ( ncomps ) )
 dhcalcdx = 0.0

!! Determine variable indices and unpack

 hlpos = iv + ncomps + 2
 Tpos  = iv + ncomps + 3
 Ppos  = iv + ncomps + 4
 xbase = iv + ncomps + 4
 idbase = iv + 1

 hl = fms_vars(fms_eqn(hlpos))%value
 T  = fms_vars(fms_eqn(Tpos))%value
 P  = fms_vars(fms_eqn(Ppos))%value

!! Convert to format required by external package

 do comp = 1, ncomps

    compindex(comp) = nint(fms_vars(fms_eqn(idbase + comp))%value)
    x(comp) = fms_vars(fms_eqn(xbase  + comp))%value

 end do

!liquid stream

 phase = 'L'

! subroutine written by Bill Morton called to calculate enthalpy of a
! liquid stream

 call enthphase( phase, ncomps, compindex, T, P, x, hcalc, &
                              dhcalcdt, dhcalcdP, dhcalcdx)

!! Transfer results to JFMS Jacobian etc
!! storage (no need for unit conversion)

 r = r + 1

 if ( jc ) then

    !!! derivative of liquid enthalpy function w.r.t T
```

```
      nj = nj + 1
      col = fms_eqn(Tpos)
      val = -dhcalcdt
      elt = 2
      call  jac_lists ( jpt , nj , r , col , val , elt )

      !!! derivative of liquid enthalpy function w.r.t P

      nj = nj + 1
      col = fms_eqn(Ppos)
      val = -dhcalcdP
      elt = 2
      call  jac_lists ( jpt , nj , r , col , val , elt )

      !!! derivative of liquid enthalpy function w.r.t h

      nj = nj + 1
      col = fms_eqn(hlpos)
      val = 1.0
      elt = 1
      call  jac_lists ( jpt , nj , r , col , val , elt )

      do comp = 1, ncomps

         xpos = xbase + comp

         !!! derivative of liquid enthalpy function w.r.t x

         nj = nj + 1
         col = fms_eqn(xpos)
         val = -dhcalcdx(comp)
         elt = 2
         call  jac_lists ( jpt , nj , r , col , val , elt )

      end do

   else
   end if

   if ( rc ) then

      val = hl - hcalc
      sv =abs ( fms_vars(fms_eqn(hlpos))%scale )

      call rhs_list (  jpt , r ,  val , sv )

   else
```

```
 end if

 if ( hc ) then
 else
 end if

!! Deallocate temporary storage

 deallocate ( compindex )
 deallocate ( x )
 deallocate ( dhcalcdx )

end subroutine liquid_enth
```

Appendix C

Adding Applications

This chapter details the steps required to add applications to the environment. As currently set-up, changes must be made both in the Java and application server code. The chapter is set out as follows:

- Data structures in the application server;
- Adding an application to the Java client;
- Adding an application to the Application Server;

C.1 Data Structures in the Application Server

Data structures in the application server mirror those in the core model definition as discussed in chapter 4. For clarity, the structures are repeated here in their application server format. Reference should be made to FMS/F90FMS/types.f90 for the implementation details, global structures are declared in FMS/F90FMS/global.f90 and are as shown in figure C.1.

```
MODULE global
  use types

!! Vector storing variable data
  type ( var )    ,allocatable     :: fms_vars(:)

!! Vector storing current Jacobian data
  type ( jacEl )  ,allocatable  :: fms_jac (:)

!! Vector storing current Hessian data
  type ( hessEl ),allocatable   :: fms_hess (:)

!! Vector storing current residual data
  type ( resEl ),allocatable      :: fms_res(:)

!! Vector storing equation allocation table
  type ( eat )    ,allocatable   :: fms_eat(:)

!! Vectors storing current specifications and initial state
  type ( spec ) ,allocatable      :: fms_spec(:) , iniTS(:)

!! Objective function
  type ( eat )                     :: obj

!! Vector declaring initialisation methods
  type ( eat ) , allocatable :: fms_ini(:)

!! Vector storing group table
  type (grp ) , allocatable :: fms_grp(:)

!! Vector storing current system variables
  type ( sys )                        :: fms_sys

!! Vectors storing method variable indices and active variable flags
  integer   ,allocatable     :: fms_eqn(:),fms_actV(:)

!! Identifies which structure is being initialised
!! (Internal FMS variable)
  integer           :: curArrayPos

END MODULE
```

Figure C.1: Data Structures within the Application Server

C.2 Adding an application to the Java client

The application must be added to the Model Handler Window and to the execProcess in the model building routines (files /FMS/GUI/FMSModHandler.java and /FMS/GUI/ModelBox.java respectively). Both files use the same calling structure but FMSModhandler requires the name to be added to the pull down menu as well. These routines pass the model data to the application server and activate the relevant applications driver routine. The resulting variable set is returned and the client updated accordingly.

Adding the Calling Structure

The code in figure C.2 should be inserted in the conditional block marked *//Add new applications here.* In this instance, the code adds the call to FilterSQP which is the 3rd application in the current installation, as indicated in the second line as the second argument of *dOut.exec.* This id number should be unique for each application and mirrored in the calling routine in the application server.

```
else if ( selected.equals ( "FilterSQP" ) ) {
 temp = dOut.exec(temp.length , 3);
 for ( k = 0 ; k < temp.length ; k++ )
  prob.getPVars().elAt(k).setVal(temp[k]);
}
```

Figure C.2: Code required to add an application to the Java Client

Adding the Application to the Model Handler Window Menu

Within /FMS/GUI/FMSModHandler.java, the new application should also be added to the pull down menu. The code required takes the form *menu.addItem ("***Application Name***")* and should be added to the existing block.

C.3 Adding an application to the Application Server

An application is added to the calling routine in the application server by adding a call to its' driver routine under the appropriate application id number (as specified in the java client) in the file /FMS/Comms/f90exec.f90 case block and adding the required module(s) at the top of the file. The Makefile and work.pcl file in the Comms directory should also be amended to include the new files and their home directory.

The application (or it's driver) should control which methods are to be executed. This is achieved by setting the active/inactive flag for the method within fms_eat, the equation allocation table.

C.3.1 Application Drivers

The application driver is used to convert the JFMS Application Server Data into the format required by the application itself. Once the data is correctly formatted it is passed to the application for processing and control passes to the applications' own interface and internal routines. Once the application terminates the output should be converted back to the JFMS format and the JFMS variable values updated to equal the output of the application.

C.3.2 Accessing JFMS Methods from within the Application

The module /FMS/F90FMS/jac_hess.f90 contains a routine called **fms_build_jh**. This routine processes the current active methods and updates the appropriate JFMS structures. It provides a wrapper to the method library defined in chapter B and takes the following form:

call fms_build_jh (cj , ch , cr , co , eval)

where:

- cj - logical variable controlling whether Jacobian is to be calculated;
- ch - logical variable controlling whether Hessian is to be calculated;
- cr - logical variable controlling whether residual is to be calculated;
- co - logical variable controlling whether objective is to be calculated;
- eval - real value

Before calling fms_build_jh it is important to update the JFMS variable set to the values currently held in the application. Assuming that the application stores current variable values in a real vector x of length n, n = number of variables, this is done using the following code:

fms_vars (1 : n)%value = x (1 : n)

Appendix D

Extraction of Equation and Variable Subsets

This section details the steps required to extract equation and variable subsets from the model for subsequent processing. This allows model decomposition against user specified criteria such as entity by entity.

All data required is stored in the Fortran90 data structures. A standard routine (GrpVESubset in module jac_hess) is provided to extract the variables and equations belonging to a specific entity. This routine can be run several times in order to construct a block of equations and variables from several entities. The user should call this function based on the selection criteria they have chosen to use.

Should the user wish to extract other subsets of the model, not based on a unit by unit decomposition it would be necessary to produce a Fortran90 routine to achieve this. GrpVESubset should be referred to for details, but in essence the extraction is achieved by identifying the equations that are to be called and setting their active flag to 1 (the others should be 0). As illustrated in GrpVESubset, the active variable set can be derived from the Jacobian output from a test run.

Care must be taken when linking applications to a model that will be decomposed using such a method. The variable set in the application must be linked to the fms_vars as

```
do j = 1 , size ( fms_vars )
   if ( fms_vars(j)%active ) then
     fms_vars(j)%value = x ( i )
     i = i + 1
   else
   end if
   if ( i > nvars ) exit
end do
```

Figure D.1: Mapping Active Variables to the Global Variable Set

shown in Figure D.1 as the equation function calls are based on the variables id number in the global variable set rather than the local one.

This is in effect the reverse of the process used to determine which variables are active.

References

Abbott, K. A. (1996). *Very large scale modeling.* PhD thesis, EDRC, CMU.

Abbott, K. A., Allan, B. A., and Westerberg, A. W. (1997). Global preordering for Newton equations using model hierarchy. *AIChE Journal*, 43(12):3193–3204.

Allan, B. A. (1998). *A more reusable modeling system.* PhD thesis, EDRC, CMU.

Allan, B. A. and Westerberg, A. W. (1999). Anonymous class in declarative process modeling. *Industrial & Engineering Chemistry Research*, 38(3):692–704.

Amarger, R. J., Biegler, L. T., and Grossmann, I. E. (1992). An automated modeling and reformulation system for design optimization. *Computers & Chemical Engineering*, 16(32):623–636.

ASPEN (1992). *Speedup Manual.* ASPEN Technology.

ASPEN (2000). *ASPEN: An advanced system for process engineering.* ASPEN Technology.

Ballinger, G. H., Alcantara, R. B., Costello, D., Fraga, E. S., Krabbe, J., Lababidi, H., Laing, D. M., McKinnel, R. C., Ponton, J. W., Skilling, N., and Spenceley, M. W. (1993). epee: A process engineering environment. *Computers & Chemical Engineering*, 18(Suppl):283–287.

Banares-Alcantara, R., Lababidi, H. M. S., Ballinger, G., and King, J. M. P. (1994). KBDS: A support system for chemical engineering conceptual design. *Proc. ESCAPE-4, IChemE*, pages 419–426.

Barton, P. I. (1992). The modelling and simulation of combined discrete/continuous processes. *Ph.D. Thesis, Imperial College of Science, Technology and Medicine.*

Bieszczad, J., Koulouris, A., and Stephanopoulos, G. (1999). MODEL.LA: A phenomenon-based modeling environment for computer-aided process design. *Technical Program & Preprints: FOCAPD 1999.*

Brooke, A., Kendrick, D., and Meeraus, A. (1988). *GAMS - A users guide.*

Cochrane, S. (1999). Initialisation methods for distillation columns. Technical report, School of Chemical Engineering, Edinburgh University.

Croner, A., Holl, P., Marquardt, W., and Gilles, E. D. (1990). DIVA - An open architecture for dynamic simulation. *Computers & Chemical Engineering*, 14(11):1289–1295.

Douglas, J. M. (1988). Conceptual design of chemical processes. *McGraw Hill.*

Farrell, M. (1998). MICE: Model Information Checking Engine. Technical report, School of Chemical Engineering, Edinburgh University.

Felton, J. (1996). Model formulation. Master's thesis, School of Chemical Engineering.

Fletcher, R. and Leyffer, S. (1998). *User manual for filterSQP.*

Fraga, E. S. and McKinnon, K. I. M. (1994). CHiPS: A process synthesis package. *Chemical Engineering Research and Design*, 72 A3:389–394.

Griffiths, M. (1997). A review of the Flexible Modelling System (FMS). Technical report, School of Chemical Engineering, Edinburgh University.

Hyprotech (1995). *HYSYS user manual.* Hyprotech Ltd.

Iordanis (1999). MAPLE generation of FMS equation routines. Technical report, School of Chemical Engineering, Edinburgh University.

Lewis, W. K. and Matheson, G. L. (1932). Studies in distillation. *Industrial Engineering Chemistry*, 24:494.

Locke, M. H. and Westerberg, A. W. (1983). The ASCEND-II system - A flowsheeting application of a successive quadratic-programming methodology. *Computers & Chemical Engineering*, 7(5):615–630.

Lorek, H. and Sonnenschein, M. (1999). Modelling and simulation software to support individual-based ecological modelling. *Ecological Modelling*, 115(41):199–216.

Marquardt, W. (1994). Computer-aided generation of chemical engineering process models. *International Chemical Engineering*, 34(1):28–45.

Marquardt, W. (1996). Trends in computer-aided process modeling. *Computers & Chemical Engineering*, 20(6/7):591–609.

Michalewicz, Z. (1994). *Genetic Alogorithms + Data Structures = Evolution Programs.* Springer-Verlag.

Mitchell, D. R. and Morton, W. (1996). Modelling techniques for large scale process engineering problems. Technical report, School of Chemical Engineering, Edinburgh University.

Morton, W. (1996). On the robust formulation of heat exchanger models. *ECOSSE Technical Report, Edinburgh University.*

Morton, W. and Collingwood, C. (1998). An equation analyser for process models. *Computers & Chemical Engineering*, 22(4/5):571–585.

Murtaugh, B. H. and Saunders, M. A. (1985). *MINOS user's guide.*

Oh, M. and Pantelides, C. C. (1996). A modelling and simulation language for combined lumped and distributed parameter systems. *Computers & Chemical Engineering*, 20(19):611–633.

Perkins, J. D., Sargent, R. W. H., Vazquez-Roman, R., and Cho, J. H. (1996). Computer generation of process models. *Computers & Chemical Engineering*, 20(6/7):635–639.

Piela, P. C. (1989). *ASCEND: An object-orientated computer environment for modeling and analysis.* PhD thesis, EDRC, CMU.

Piela, P. C., Epperly, T. G., Westerberg, K. M., and Westerberg, A. W. (1991). ASCEND - An object-orientated computer environment for modeling and analysis - The modeling language. *Computers & Chemical Engineering*, 15(1):53–72.

Ricoramirez, V., Allan, B. A., and Westerberg, A. W. (1999a). Conditional modeling. 1. Requirements for an equation-based environment. *Industrial & Engineering Chemistry Research*, 38(2):519–530.

Ricoramirez, V., Allan, B. A., and Westerberg, A. W. (1999b). Conditional modeling. 2. Solving using complimentarity and boundary-crossing formulations. *Industrial & Engineering Chemistry Research*, 38(2):531–553.

Rodriguez-Toral, M. A. (1999). *Synthesis and optimisation of large-scale utility systems.* PhD thesis, School of Chemical Engineering, Edinburgh University.

Rodriguez-Toral, M. A., Morton, W., and Mitchell, D. R. (1999a). The use of new SQP methods for the optimization of utility systems. *Submitted to Computers & Chemical Engineering.*

Rodriguez-Toral, M. A., Morton, W., and Mitchell, D. R. (1999b). Using new packages for modelling, equation-orientated simulation and optimization of a cogeneration plant. *Submitted to Computers & Chemical Engineering.*

SACDA (1993). *MASSBAL User Manual.* SACDA Inc, London, Ontario, Canada.

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

Stephanopoulos, G., Henning, G., and Leone, H. (1990a). MODEL.LA. A modeling language for process engineering-1. The formal framework. *Computers & Chemical Engineering*, 14(8):813–846.

Stephanopoulos, G., Henning, G., and Leone, H. (1990b). MODEL.LA. A modeling language for process engineering-2. Mulitfaceted modeling of processing systems. *Computers & Chemical Engineering*, 14(8):847–869.

Stephanopoulos, G., Johnston, J., Kriticos, T., Lakshmanan, R., and Mavrovouniotis, M. (1996). Design-kit: An object-orientated environment for process engineering. *Computers & Chemical Engineering*, 20(6/7):591–609.

Verbeek, V. (1999). Evaluation of novel optimisation methods. Technical report, School of Chemical Engineering, Edinburgh University.

Westerberg, A. W. (1997). The creation of computer-based environments to support design. *Abstracts of Papers of the American Chemical Society*, 213.

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

Zoppke-Donaldson, C. (1995). *A tolerance-tube approach to sequential quadratic programming with applications.* PhD thesis, Department of Mathematics, University of Dundee.