

Chronicling Process Model Construction using World Wide Web Technology

Samuel Ross Andrews

Thesis presented for the Degree of Doctor of Philosophy

University of Edinburgh

2001



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 explicitly stated 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 the following people and organisations for their assistance throughout the course of my PhD:

- My supervisor, Professor Jack Ponton, for his support and guidance over the years.
- E. I. DuPont de Nemours and the Engineering and Physical Sciences Research Council for providing the funding for this project, and Edinburgh University for providing the infrastructure.
- My friends and colleagues in the department of chemical engineering for their advice and support, and for helping to keep the working day fun.
- My family for their sustained interest and support over the years.
- Finally to my girlfriend Catherine for perservering with me for the final months of the PhD, and without whom my life would be much less complete.

Abstract

In developing and constructing process models, a large amount of data are generated. This needs to be stored as information by providing context (e.g. description and units), such that it may be accessed and understood by both human users and computer tools. To be useful, information sets must be chronicled by recording information such as by whom it was generated, when, and using what tool. Links should also be maintained between related sets of information to document the relationship.

The World Wide Web (www) facilitates the provision of links between documents. It may, amongst other things, be thought of as a globally addressable file system. Accessed by a graphical tool, the browser, it can provide a common user interface to access and interact with documents across disparate operating systems. Documents on the www include hypertext links, which enable one document to link to another.

This thesis describes an object-oriented system which makes use of the www to chronicle process model construction. The object-oriented paradigm has been used to provide a convenient mechanism for encapsulating data in a structured framework. As well as containing the object data, objects also contain information on who created the object, at what time, etc. This information may be used to generate a browsable history of object creation.

Objects are stored as www documents, and may be created, viewed and manipulated using standard browsers. An application programmer's interface has been written which enables the information to be manipulated via Fortran programs. Objects have been developed for standard process engineering entities such as streams, mixtures, components, process topologies, etc. Once created, these may be used to generate simulation models in a variety of formats, e.g. ASPEN, spreadsheet model, etc. Conversely, process objects may be generated from ASPEN models. Four case studies have been included, showing various applications of the system.

In conclusion, the www provides a suitable environment for implementing such a system, due to its ease of use, and the fact that it provides both a user interface and enables remote access to the system.

Contents

1	Introduction	1
1.1	The World Wide Web	2
1.2	Thesis Outline	4
2	Technology Review	5
2.1	Introduction	5
2.2	Process Engineering Information Management Systems	5
2.2.1	Requirements of a Process Engineering System	6
2.2.2	Object Orientation	7
2.2.3	MDOOM	9
2.2.4	N-dim	11
2.2.5	Epée	13
2.2.6	Comparison Against Requirements	16
2.3	The World Wide Web	20

- 2.3.1 User Interaction 23
 - 2.4 WWW-based Systems 25
 - 2.4.1 Use of the WWW in Process Engineering Information Management Systems 25
 - 2.4.2 Computer Supported Cooperative Work 26
 - 2.4.3 CSCW Implementation on the WWW 29
 - 2.5 Discussion 30
 - 2.6 Summary 33
- 3 Design of a WWW Based Model Building System 34
 - 3.1 Introduction 34
 - 3.2 Object Structure 35
 - 3.2.1 Data Slots 35
 - 3.2.2 Logistical Slots 36
 - 3.2.3 Methods 37
 - 3.3 Object Creation and Manipulation 39
 - 3.3.1 Entering the System 40
 - 3.3.2 Method Application 45
 - 3.3.3 Object Naming 46

3.4	Other Selection Options	48
3.4.1	Mail all objects to entered email address	48
3.4.2	Create a new object	48
3.4.3	Finish the session	49
3.4.4	Obtain an object from a URL	49
3.4.5	Create a new custom class	50
3.5	History Options	52
3.6	Summary	55
4	Process Engineering Objects	56
4.1	Introduction	56
4.2	Component	57
4.2.1	Update the object to register any manual changes	58
4.2.2	Add a slot to the object	58
4.2.3	Remove a user defined slot from the object	59
4.2.4	Get properties from the databank	59
4.2.5	Estimate properties using group contribution method	59
4.2.6	Create a spreadsheet containing the object data	61
4.3	Mixture	62

4.3.1	Add component(s) to the object	62
4.3.2	Remove components(s) from the object	63
4.3.3	Create a spreadsheet containing component data	63
4.3.4	View VLE data on selected components	63
4.4	Stream	64
4.4.1	Add component(s) to the object	66
4.4.2	Remove component(s) from the object	66
4.4.3	Create a spreadsheet containing the object data	66
4.4.4	Add a new phase to the stream	66
4.5	Reaction	67
4.5.1	Add component(s) to the reaction	68
4.5.2	Calculate reaction properties	68
4.5.3	Clear other data slots	69
4.6	Unit	69
4.6.1	Update the object to register any manual changes	70
4.6.2	Add a slot to the object	70
4.6.3	Remove a user defined slot from the object	71
4.6.4	Create an ASPEN model of the unit	71

4.6.5	Use the latest version of referenced objects	71
4.7	Topology	72
4.7.1	Update the object to register any manual changes	73
4.7.2	Add a slot to the object	77
4.7.3	Remove a user defined slot from the object	77
4.7.4	Decompose single unit into two or more units	77
4.7.5	Separate units into different topology objects	78
4.7.6	Combine topology objects together	78
4.7.7	Change stream/unit target	79
4.7.8	Use latest versions of objects	79
4.7.9	Create a spreadsheet model	80
4.7.10	Return data from a solved spreadsheet model	82
4.7.11	Create an ASPEN model	83
4.7.12	Create a HYSYS model	84
4.7.13	Get data from an ASPEN model	85
4.8	Equation	87
4.8.1	Solve the equations	89
4.8.2	Edit the equations	90

4.8.3	Delete some equations	90
4.8.4	Add new equations/variables	90
5	Application Programmer's Interface	92
5.1	Introduction	92
5.2	Implementation	93
5.3	Fortran Structures	93
5.4	Fortran Subroutines	95
5.4.1	Overloaded Operators	96
5.5	Example Program	97
5.6	Summary	101
6	Case Studies	102
6.1	The Creation of a Topology Object and an ASPEN model	102
6.1.1	Plant Description	103
6.1.2	Topology Object Construction	104
6.1.3	Parameter Entry	106
6.1.4	Model Creation	108
6.1.5	Discussion	109

6.2	The Creation of a Spreadsheet Model from an ASPEN Model	109
6.2.1	Cyclohexane Manufacture	110
6.2.2	Topology and Associated Objects	111
6.2.3	Spreadsheet Model	111
6.2.4	Discussion	111
6.3	The Manipulation of ASPEN from a Fortran Program	114
6.3.1	Implementation	114
6.3.2	Discussion	116
6.4	A Fortran Program with a WWW Interface	117
6.4.1	Reactor Description	117
6.4.2	Custom Object Class	118
6.4.3	Method Application	120
6.4.4	Discussion	122
6.5	Summary	123
7	Discussion	125
7.1	Process Engineering Information Management System Criteria	125
7.1.1	Efficiency	126
7.1.2	Persistent Data Storage	128

7.1.3	User Extensible	129
7.1.4	Application Integration	130
7.1.5	Audit Trail Maintenance	130
7.2	Use of WWW Technology	131
7.2.1	Client-Server Processing Distribution	132
7.3	API discussion	133
7.4	Model Construction	134
7.5	Summary	135
8	Conclusions and Future Work	137
8.1	Conclusions	137
8.2	Future Work	139
8.2.1	Object Definitions	139
8.2.2	File Based Approach	140
8.2.3	Other Problems	141
A	Unit Description Parsing	144
A.1	Distillation Column	146
A.2	Splitter	147

A.3	Reactor	148
A.4	Flash	150
A.5	Mixer	151
A.6	Contactor, Stripper, Absorber	151
A.7	Heater, Cooler	152
A.8	Vaporiser, Condenser	153
B	Generation of Equations for Unit Types	154
B.1	Feed	154
B.2	Mixer	155
B.3	Reactor	155
B.4	Splitter	156
B.5	Separator	157
B.6	Heater, Cooler, Vaporiser, Condenser	157
B.7	Contactor, Stripper, Absorber	158
C	Class Files	159
C.1	Class File Structure	159
C.2	Base Object	162

C.3 Component	163
C.4 Mixture	164
C.5 Stream	164
C.6 Reaction	165
C.7 Unit	165
C.8 Topology	166
C.9 Equation	166
D Object Data Storage	168
D.1 Introduction	168
D.2 HTML Files	168
D.3 XML Files	169
D.4 Document Type Definition	171
D.5 Example Stream XML File	172
E Miscellaneous Files	175
E.1 VLE Data Generation Input File	175
F Recognised Units	176
G API data	178

G.1	API Fortran Structures	178
G.1.1	Parameters	178
G.1.2	slot	179
G.1.3	multiple	179
G.1.4	nest	180
G.1.5	Object Structures	180
G.2	API Fortran Subroutines	182
G.2.1	www_setup_api	182
G.2.2	www_apply_method	183
G.2.3	www_write_html	183
G.2.4	www_get_obj_from_html	184
G.2.5	www_initialise	184
G.2.6	www_set_slot	185
G.2.7	www_write_xml	185
G.2.8	www_add_slot	186
G.2.9	www_remove_slot	186
G.2.10	www_add_comp	187
G.2.11	www_get_mult_slot_attribute	187

G.3	Method Parameters	188
H	Parameter Extraction for Simulation Models	192
H.1	Component Names	192
H.1.1	ASPEN	192
H.1.2	HYSYS	193
H.2	Unit Connectivity	193
H.3	Feed Streams	193
H.4	Unit Parameters	193
H.4.1	Mixer	194
H.4.2	Splitter	194
H.4.3	Reactor	194
H.4.4	Separator	195
H.4.5	Heater	196
H.4.6	Cooler	196
H.4.7	Vaporiser	196
H.4.8	Condenser	196
H.4.9	Contact, Stripper and Absorber	197

I	Case Study Data	198
I.1	HCN plant ASPEN model	198
I.2	Cyclohexane model data	199
I.2.1	ASPEN model input file	200
I.2.2	Model Topology	201
I.2.3	ASPEN model solutions	202
I.2.4	Spreadsheet Model	204
I.3	Distillation Column Data	205
I.3.1	ASPEN Model Input File	205
I.3.2	API program	206
I.3.3	Output of API Program	211
I.4	HCN Reactor Data	212
I.4.1	Reactor Class File	212
I.4.2	Reactor Custom Method CGI Script	212
	References	222

List of Figures

1.1	Atomic and Higher Level Entities	1
2.1	Inheritance in object-oriented systems	8
2.2	Operation of a Client - Server System	9
2.3	Object evolution in n -dim	12
2.4	Processing Form Data	24
2.5	Browser History Illustration	28
3.1	The data slots of a partially complete component object	36
3.2	The logistical slots of an object	37
3.3	The methods of a component object	38
3.4	The help function for object methods	38
3.5	Starting a Session	41
3.6	Selecting component names	42
3.7	Selecting an approximate structure	44

3.8	Applying a method to an object	47
3.9	Object Inheritance	50
3.10	Applying a custom method	53
3.11	A history tree for an object family	54
3.12	A manifest table for mixture objects	54
3.13	The logfile of object creation	55
4.1	Adding a slot to an object	59
4.2	Entering the molecular formula using the molecular typewriter	60
4.3	Returning component properties from estimation method	61
4.4	The output from calculating VLE data on Benzene and Toluene at 20 bar.	64
4.5	Data slots for a stream object	65
4.6	A Sample History Tree	72
4.7	The decomposition of a black box model	78
4.8	The frames version of an equation object	88
5.1	Method execution on the WWW server from remote clients	94
6.1	Diagram of a simple hydrogen cyanide plant	103
6.2	A topology with one recycle stream with no purge	105

6.3	Aspen model of a cyclohexane plant	110
6.4	History of topology creation for API program	116
6.5	The methods of a reactor object	122
8.1	Three tier architecture for WWW applications	140
B.1	A mixer unit	155
B.2	A reactor unit	155
B.3	A splitter unit	156
B.4	A separator unit	157
B.5	A heater unit	157
B.6	A contactor unit	158
I.1	Aspen model of a cyclohexane plant	200

List of Tables

4.1	Operation names recognised by parsing	74
6.1	Reaction coefficients for the reaction between NH_3 and CH_4	107
6.2	Stream flows from the two cyclohexane plant models at a reduced benzene conversion of 98%	113
A.1	Keywords recognised by parsing and indicated unit functions	144
G.1	Method codes and parameters for component objects	188
G.2	Method codes and parameters for mixture objects	189
G.3	Method codes and parameters for stream objects	189
G.4	Method codes and parameters for reaction objects	190
G.5	Method codes and parameters for unit objects	190
G.6	Method codes and parameters for topology objects	191

Chapter 1

Introduction

Design is: “the creative activity whereby we generate new ideas and then translate them into equipment and processes for producing new materials or for significantly upgrading the value of existing materials” (Douglas [18]). In generating and translating these new ideas, this activity also generates a large amount of data, e.g. process synthesis, process simulation. The data generated need to be recorded as information. Data are turned into information by adding context, e.g. 315.6 is an item of data, but “input stream temperature of 315.6 K” is information. The information needs to be stored in an organised way such that it can be accessed and understood by both human users, and computer tools. To aid in this understanding the information should also maintain a “sense” about itself, where it came from, how much trust can be put in it, etc. This takes place at two levels: for individual atomic entities, and for higher level entities.

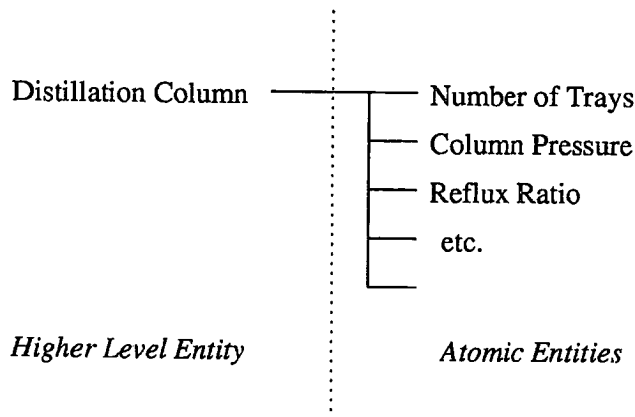


Figure 1.1: Atomic and Higher Level Entities

For individual atomic entities such as number of trays in a distillation column, column pressure, etc., some basic facets need to be recorded. These include things such as: the name of the entity, its value, units, and how much trust may be put in the value of the entity, etc. Higher level entities are composed of several of the individual atomic entities, e.g. the examples given above of atomic entities would be contained in a distillation column entity (see figure 1.1). The facets recorded for individual atomic entities apply solely to those atomic entities. However the facets recorded for the higher level entity refer equally to each and every one of the individual atomic entities which comprise the higher level entity. These facets include who created the entity, using what, at what time, etc. This is known here as *chronicling*.

The Oxford English Dictionary [1] defines the verb “to chronicle” as: “to record (events) in the order of their occurrence”. Process design is by its very nature a series of activities. A product needs to be manufactured. Synthesis suggests several routes. Several of these routes are evaluated, perhaps using simulations, and refined. Finally one is chosen as best fits the criteria. *Chronicling* these tasks means that as well as recording the information generated at each step, information *about* the information is also recorded, e.g. who generated the information, etc.

As specific sets of design information will be related to other sets of information, especially those generated during refining of models, there needs to be some method of maintaining the relationship between these different sets of information, i.e. that one set was generated by performing an operation upon another set of information. One commonly used tool which facilitates the provision of links between documents is the World Wide Web (www).

1.1 The World Wide Web

At its simplest, the World Wide Web (www) may be thought of as a globally accessible file system. It is normally accessed using a graphical user interface, the “browser”, in which the user may view documents. Documents on the WWW are written in HyperText

Markup Language (HTML). This is a formatting language, which enables the user to include images, multimedia, and text in a document. More importantly it also includes the concept of hypertext links, or hyperlinks. These enable another document to be “linked” from the current document, with the link being contained within a word or phrase, or even in an image in the document. Activating this link, by “clicking” on the words or image loads up the new document into the browser. Several different types of files may be viewed directly in the browser. For other file types, the browser may be configured to use a local tool to view them.

Why use the www in this project? Giradot [23] lists a number of advantages of the www:

- it provides a universal Graphical User Interface (GUI).
- it provides a universal communication protocol.
- it is here for a long time.
- it is an open system.
- none of the technology is proprietary, and most is free.

Due to the popularity of the www it can safely be assumed that for all but the most obscure operating system / hardware platform combination, there will almost certainly be a browser. This removes one of the more complicated tasks in most software projects, that of designing a GUI and porting it to several platforms. Almost as important for a distributed application is the communication protocols between the different elements of the system. Again this is provided by using the www.

It was the intention of this work to develop a user extensible system, making use of the www, to chronicle model construction. Previous efforts at developing such a system have spent a large amount of effort in duplicating facilities which are now made available by the www for little or no work.

1.2 Thesis Outline

Chapter two discusses the WWW in more detail. More specifically it focuses on aspects of WWW technology. It also examines several fields relevant to this project: process engineering information management systems and computer supported cooperative work. Chapters three and four describe a system intended to chronicle model construction, accessed via WWW technology. Chapter five shows how the system may be accessed using Fortran programs. Chapter six describes several case studies. Chapter seven is a discussion of the system features. Finally, chapter eight ends the main body of the thesis with a set of conclusions and some ideas for future work.

Chapter 2

Technology Review

2.1 Introduction

The introduction chapter detailed the reasons behind chronicling process model construction. This chapter discusses prior research and technology relevant to this project. The first section reviews process engineering information management tools, examining three of these in detail against a list of requirements for such systems. The next section discusses the World Wide Web (www), in particular the technology aspect. We then examine a field with similar aims to the proposed project, but which has made extensive use of the www, namely Computer Supported Cooperative Work (CSCW). Finally the requirements of a process engineering information management system are discussed to see how many are met by default by using a system based on the www, and how the remaining requirements may be fulfilled by a www based system.

2.2 Process Engineering Information Management Systems

This section examines process engineering information management systems. Many systems have been developed to manage process engineering information, e.g. Design Kit [44], Plantool [26], Version Administration System [24]. This section will examine

three systems in detail: MDOOM [33], *n*-dim [54] and *épée* [15]. These systems are reviewed because of the built-in structures used to store data. MDOOM has an exhaustive definition of all possible structures and data fields considered necessary in process engineering. The structures may not be extended by the user. N-dim has no predefined structures present, and any used must be specified by the user. *Epée* lies in between these two systems, in that there are a series of process engineering structures, but they may be extended by the user.

First of all the requirements of a process engineering management system are discussed. Then as most of the systems mentioned in the previous paragraph claim to be object-oriented, the object-oriented paradigm is discussed. After this section the three systems (*n*-dim, MDOOM and *épée*) are reviewed in detail. For each system, a brief summary is given of the main features. Next there is a discussion of the system, analysing the strengths and weaknesses of that particular system. Finally once each system has been examined in detail, they are all then compared against the list of requirements of a process engineering information management system.

2.2.1 Requirements of a Process Engineering System

A list of requirements for a process engineering system has been given by Fraga et al. [20]:

- **Distributed System** - The system must be inherently distributed in nature. It should make efficient use of the resources that a network can provide, e.g. remote access, use of fast machines for compute intensive tasks, etc.
- **Efficiency** - Efficiency is crucial. If the overheads are large, users will be reluctant to use it. This applies both to the time to access the system, the time to run simple tasks, and also the time taken to learn the system itself.
- **Persistent Data Storage** - A mechanism for creating persistent data structures is necessary. It must be possible to access data over a period of time.
- **User Extensible** - The system must be easily extensible by the user. It should not require computer experts.

- **Simultaneous Platform Support** - The system should support a variety of computer platforms simultaneously.
- **Application Integration** - It must be possible to develop new applications easily and using standard computer languages.

To these requirements another must be added, that of recording an audit trail of the interaction between users and the system. Although not explicitly stated as a requirement, it was an important feature of the subsequent system developed by Fraga et al. [20] to satisfy these requirements, *épée*.

The original requirements, together with the audit trail maintenance requirement, make a good starting point for the aims that a process engineering information management system should fulfil.

2.2.2 Object Orientation

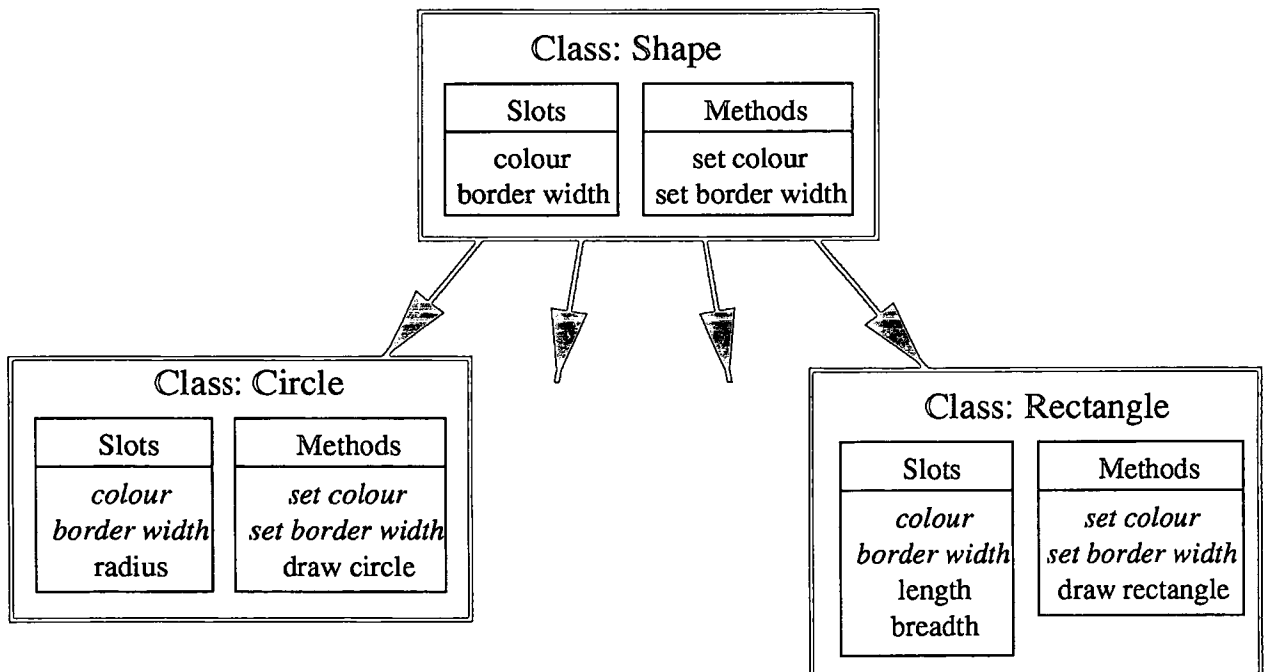
“Data must be organised in logical units that reflect their meaning and intended use” (Costello et al. [15]). All of the systems which will be discussed in the subsequent sections describe themselves as object-oriented, and use objects as “logical units” to organise the data. But what is an object, or what is object orientation?

One definition of an object is: “an object is an encapsulation of state (data values) and behaviour (operations)” (Budd [13]). As Budd also notes there are many definitions of what is an object, and what object orientation actually is. These definitions vary even on the terms used to describe the various concepts. This section describes some of the common facets of object orientation.

One of the more frequent uses of objects is to encapsulate data in a single entity. This is where the data are grouped together in a structured manner, stored in *slots*. The slots can be said to have *attributes*, e.g. slot name, slot value, etc. At this point the structure is identical to an abstract data type. To turn an abstract data type into an object requires the addition of procedures, or *methods*, which act on this data. Every

object is an instantiation of a *class*. The object class may be considered to be the list of data slots and methods.

To be considered an object-oriented system, rather than just an object-based system, the object classes must support *inheritance*. Inheritance is where the object classes are arranged in a hierarchy, with objects lower down in the hierarchy inheriting both slots and methods from the classes higher up in the hierarchy without explicitly restating either the slots or methods. Figure 2.1 illustrates this with an example. The superclass “Shape” has slots “colour” and “border width”, and methods “set colour”, and “set border width”. Two classes inherit from the shape class: “circle” and “rectangle”. They both define further slots and methods. The circle class defines a slot “radius”, and a method “draw circle”. Each of these classes inherit the slots and methods of the superclass “shape”. Thus the full list of slots for the circle class is: “colour”, “border width” and “radius”. Similarly the methods for the circle class are: “set colour”, “set border width” and “draw circle”. Similarly the methods for the circle class are: “set colour”, “set border width” and “draw circle”.



Slots and methods defined in classes are written in ordinary type.
Those inherited are written in *italic type*

Figure 2.1: Inheritance in object-oriented systems

2.2.3 MDOOM

Description

MDOOM stands for Multi-Dimensional Object-Oriented information Model, and is used to model the entire life cycle of a plant, from original design, through operation to decommissioning. It is based on the STEP (STandard for the Exchange of Product model data) data model.

STEP is designed to enable the generic storing, accessing and transferring of data among and between different sectors. It is an ISO standard, ISO 10303, which is split into many subparts. One section of the standard, which contains models drawn up by individual sectors, are the Application Protocols (APs). These contain an exhaustive definition of all of the information needed by that particular sector. There are three APs which model chemical engineering data:

- AP221 [30] - Functional data and schematic representation
- AP227 [43] - Spatial representation for process plants
- AP231 [41] - Process engineering data: Process design and process specification of major equipment

Of these, probably AP231 is the most relevant.

MDOOM is a client-server object-oriented system. Clients and servers are both software programs and are normally, though not necessarily, run on different machines. The client sends a request to a server which responds to that request, e.g. by running a program, returning a file, etc.

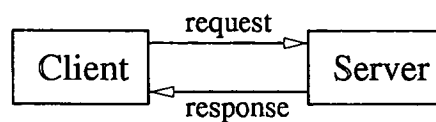


Figure 2.2: Operation of a Client - Server System

A generic object has been defined, and all other objects inherit from it. Objects have

been exhaustively defined for process entities, based on the STEP model, and are stored in a database in STEP format. These objects do not appear to be able to be extended by the user.

According to Lu et al. [33] each object “maintains a history record, including object creation and termination”. From this it would seem that objects may be removed.

The multi-dimensional facet of MDOOM is the fact that the objects can be looked at with a number of different views. Lu et al. [33] quote three views: *process*, *facilities* and *equipment*, whilst Lu et al. [34] add another view, that of the *human team*. These views are defined as follows:

- **process** - a process produces products using facilities. Attributes include operating conditions.
- **facilities** - these are logical abstractions of process equipment which are in service, e.g. for columns, the number of trays, type of trays, etc.
- **equipment** - this view handles the real physical information such as parts, assembly, structure, etc.
- **human team** - composed of the design engineers and the operating team.

Objects may be examined using a hyperviewer. This does not view the actual objects directly. Instead it views “hyperobjects”, which are hyperlinked to the actual objects inside the database.

The hyperviewer is integrated with the www and can be used to browse both local and remote data. In designing this, Lu et al. [32] note that the “end user requires to: access system data to check current design and operation status as well as design and operation history; carry out design and operational activities by having a set of support tools.”

Discussion

There are some advantages to exhaustively defining an object hierarchy, although only if the definition is complete. If the user wishes to extend an object by adding their own definitions, this should be possible. It is not clear here how this may be done, or if it is even possible.

Having multi-dimensional views of the objects helps to cut down on extraneous information when focusing on a particular aspect. However if this is necessary, it suggests that the objects may be too complex, or have too many attributes. It should be possible to view and understand all object attributes without need of a filter.

Using STEP to store the model also has its disadvantages. As a STEP model sets out to cover every piece of information, it is by necessity exhaustive, and can get quite complex very easily. AP231 [41] is over two and a half thousand pages long, and that is just for one AP. Teague and Baldwin [47] note that the “learning curve on STEP methodology is quite substantial, estimated to be on the order of a work-year”. However, perhaps the biggest drawback is that it cannot be extended, other than through a committee revision. A user cannot add another data type that he considers necessary to model a particular item and still correspond to the standard.

2.2.4 N-dim

Description

N-dim stands for N-Dimensional Information Modelling, and is a general purpose system for information modelling, not restricted to chemical or process engineering. Its creators noted that: “engineers use a variety of media (which) run the gamut from sketches, notes, diagrams to abstract graphical objects” [45]. To cope with this they: “envisioned the kind of infrastructure ... templates for capturing, accessing, reusing any of the various types of information generated during system development in a navigable browsable information web” [16]. The result of their efforts was *n*-dim which is

designed for the “capturing, organising and sharing of data and information” [55].

The structure of n -dim is a client-server, prototype-based, object-oriented system. The difference between prototypes and classes is that if a change is made in a class definition, then that change is propagated throughout all instantiations of the class. However if a change is made in the prototype definition, then only future instantiations include this alteration. There are two different types of objects: atomic and structured. Atomic objects cannot be broken down any further, e.g. an image, a piece of text. Structured objects are composed of atomic objects. Rather than including the atomic object in the structured object, the atomic object is referenced. As a result of this, each object needs only to be stored once.

Objects are stored in databases, and are given a unique name comprising of owner, time of creation, time of publishing and the language used to create it. The modelling language defines the type of objects and links that may be put in to the object. Objects are created private, meaning only their creator can alter them. The next stage is for the object to be made public, where any authorised person may edit the object. The final stage is for the object to be published, upon which it may not be changed. Once the object has been made public, in order to make any alterations it must be copied and the copy altered. A history record of is maintained of this. Access and ownership control is present, but this is no stronger than the built-in file permissions of Unix. There is no object hierarchy, so the user may set up the object to suit their own needs.

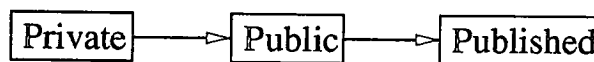


Figure 2.3: Object evolution in n -dim

Slots in n -dim can be one of three types:

- **value** - a simple string.
- **method** - a procedure to be invoked.
- **parent** - a reference to another object, from which the containing object inherits.

Westerberg et al. [54] note that adding procedures is currently difficult. The reason for

this is that the user has to program in the underlying language, stich.

All objects have intrinsic slots. These are listed by Levy et al. [31] and include who created the object, at what time, when it was last modified, etc.

In the standard visualisation tool objects appear as boxes, connected by directed arrows expressing relationships between objects, e.g. one object is “part of” another object.

Discussion

N-dim has a number of strengths. There is no predetermined set of objects, thus an experienced user can set up objects to suit their particular needs. Any object may be referenced by another object rather than included, which reduces the space needed to store objects. The evolution of objects from private to public to published allows users to develop objects in a structured manner. They can instantiate the object, then make it public for other people to comment or edit it before finally publishing it and making it immutable. Finally each object has a set of intrinsic slots to store information about the object.

However there are a number of weaknesses. The lack of any predetermined set of objects means that time is spent in setting these up. For an inexperienced user this could be quite a complicated task. Significant effort seems to have been expended in setting up the user interfaces, and the environment in general. This duplicates many of the features of the www.

2.2.5 Epée

Description

Epée [20] [15], in the words of its developers, is an “object-oriented process engineering environment” [4]. The philosophy of épée was that the focus of the environment should be on the data, rather than on the programs that manipulate the data. To facilitate

this the data are stored in objects.

Initially *épée* objects are created from template files, which contain the list of slots, and the methods that may be applied. Standard template files exist for a number of process engineering entities, such as components, streams and mixtures. The format of the template file is a standard ASCII file, and may be edited by the user.

The definition of a slot in the template file consists of: the slot name, a string describing the slot, what may be contained in the slot (integer, string, etc.), the slot dimensions and the valid ranges that a slot value may take. Each slot also has an associated status, indicating how much trust the user can put in the data. Status values of: undefined, approximation, exact, interval and calculated are mentioned by Meddes [35]. It is not clear if these status values are for display purposes only, or if they are used in some method applications.

As well as instantiating new objects from a template file, they may also be created by applying a method to an existing object. Once created, objects cannot be altered. If a change has to be made to the object data, then the object is copied and the changes are made to the copy. A history is kept of method execution. Also recorded are the method applied, the parameter settings used by the method, and the object or objects returned by the method.

All objects are stored in databases. Initially the user has a scratch database. This is private to the user, and is intended only to be a temporary working area as it is regularly purged. Once the user is satisfied with the object, it may be published to an external database, where it can be examined by other users. This external database is permanent. Each object is given a name which consists of dot separated numbers based on: the host machine's Internet address, the time and date of creation, the user's UID (the user's identification number), the process identifier of the application which created the object and a counter which is incremented each time an object is created. This ensures that the object has a globally unique identifier.

As well as using databases to store objects, databases are also used for system data.

The information for the databases is stored in plain ASCII files, which are parsed to create the database files. The system databases are:

- **slot database** - stores the definition of slots
- **template hierarchy** - records the inheritance of objects
- **method database** - the list of methods
- **application database** - details of applications

The épée architecture is a client-server system. Epée achieves simultaneous platform support by having clients available for different operating systems. These clients may connect to the same server, which could be running on a different operating system to the client. Thus the workload needed for supporting another platform is limited, as only the basic client needs to be duplicated. Clients include the Graphical User Interfaces (GUIs) used to view and manipulate objects.

Epée may also be accessed via an Application Programmer's Interface (API). This is a set of routines which enables the user to write programs in a variety of languages to manipulate objects, in much the same manner as by using the GUI.

The final mechanism for accessing épée is via a command line interface, where commands may be typed in.

External applications may be integrated into épée by using a wrapper. This translates the object into the format necessary to run the application, and then turns the application output into another object or objects.

Discussion

Epée provides a framework whereby process engineering data may be encapsulated in user extensible objects, and manipulated using a set of methods, which may be added to by the user. This gives the user a convenient template for storing the data in a structured manner. However, in order to access this data with the viewers, the

user must have an *épée* server running. The data are stored in plain ASCII files, so in principle it may be viewed simply by looking at the database file. However, this is not as convenient as using the browser, and it may be difficult if the user wishes to remove the data to another site.

Another good point of *épée* is the recording of the history of method execution. Thus if one avenue of design proves fruitless, it is easy to move back and continue along a different route.

A lot of the effort put in to *épée* seems to have been in setting up the client and server software, and the communication protocols to connect them. Significant effort also had to be expended in writing the user interfaces. Much of this could be provided by the *www*.

2.2.6 Comparison Against Requirements

Three different process engineering information management systems have been described. So how well do they meet the requirements listed in section 2.2.1? These requirements are discussed in light of the systems examined.

Distributed System

All of the systems examined deal with this requirement by using a client - server architecture. The client and the server are two programs which are normally run on different machines. The machine running the server program is usually the more powerful, and more expensive. Most of the compute intensive tasks are carried on this machine. Several clients may connect to the server. The machines running these programs are usually not as powerful as the server and are cheaper. They are slower running the more intensive tasks than the more powerful server, but are perfectly adequate for the more mundane computing tasks.

Efficiency

There are three facets to the efficiency criterion: the access time, the time to run simple tasks and the time taken to learn the system itself, including the interface. Obviously without extensive experience of a system, the first two points in the above list will be difficult to evaluate. It is assumed here that for the systems examined, these times are reasonable. If not a more powerful computer could always be used to decrease the times. However if a system is poorly designed, or has a user interface which is not easy to comprehend, then the time taken to learn the system may not be increased in this manner.

There are two aspects to learning the system: learning the user interface, and understanding the concepts behind the system.

There are a number of options for a user interface. These include:

- a Graphical User Interface (GUI), where most interaction takes place using the mouse.
- a command line interface, where commands are typed at a prompt.
- an Application Programmer's Interface (API), which enables commands to be called from a program. The API will be written for a particular language.

The GUIs tend to be the most intuitive of the interfaces. Users are comfortable with the "point and click" paradigm which has become prevalent in modern computing. However, with more experience, the command line prompt can be a more powerful tool, in that commands may be executed sequentially in a more controlled and faster manner than with an equivalent mouse driven menu. The API is useful for programmers who wish to use the system in their own programs.

All of these interfaces are used in the systems examined. *Epée* uses a command line interface, together with some graphic capabilities, and has an API. *MDOOM* has a GUI, with the graphical menus bound to commands using its API. *N-dim* also has a GUI,

and an API to connect and control the system.

All of these options take time to learn, whether it is where the commands are on the menus of the GUI or what the various commands are for the command line prompt. The API contains a number of subroutine and function calls, some of which have parameters, and it takes time to become familiar with each of these.

In understanding the concepts of the system, it is useful to use terms that are familiar to process engineers. By and large engineers know what terms such as components, mixtures and streams are and what properties they have. Both *épée* and *MDOOM* are designed for use by process engineers. They have built-in objects designed to represent various process engineering entities. As *n-dim* is intended to have a much broader outlook, not just confined to process engineering, it does not have any objects for these concepts. The user will have to set them up manually. This may be an advantage in that an experienced user will be able to create data structures specific to their needs, but it will take time for a new user to reach this level of expertise.

Persistent Data Storage

All of the systems looked at use objects to encapsulate data, which are then stored in various databases. Not all of these objects are persistent. In *épée* the objects are initially stored in a scratch database. If they are to be made persistent they must be “published” to an external database. *N-dim* has an additional stage. The objects start as private, can then be made public, and can finally be published. The effect of publishing objects means that they cannot be altered. If any changes are desired to be made the object is copied, and changes are made to the copy. In *MDOOM*, it is unclear how persistent the objects are as objects may be removed.

User Extensible

Both *épée* and *n-dim* allow the user to add their own data slots and methods to objects. Indeed, *n-dim* expects the objects to be created from scratch. However, as *MDOOM*

stores its data in STEP format it means that the objects are non-extensible. The STEP model is intended to be an exhaustive model of all the information ever required, so it could be argued that there should be no need to extend the object. However there may be occasions where the user wishes to extend the object beyond any defined object, and this should be possible from within the system.

Simultaneous Platform Support

Having a client - server architecture, as all of the systems looked at have, means that only the clients need to be rewritten to deal with different operating systems and hardware platforms. This does mean that a lot of effort is duplicated, and this work must be ongoing to write clients to cope with the new operating systems released onto the market.

Application Integration

Levy et al. [31] list three levels of tool integration:

- **Capturing input/output of decoupled tools** - At this level the tools themselves may not be invoked from the environment. The data are simply captured.
- **Encapsulation of tools** - This involves a direct translation of the input data to the required format.
- **Replication of tools** - This would only be worthwhile for simple tools.

All of the systems looked at here use the second method, that of encapsulating tools and employing translations to and from the format used by the tools and between the common format used by the system. MDOOM stores its data in STEP format, which its proponents hope will become an industry standard. Thus no translation would be necessary for STEP compliant applications.

Having an intermediate storage format has its advantages. As Ballinger et al. [5] note, it means that for n applications only $2n$ translators are necessary. If no common format were employed, it would mean having to write on the order of n^2 direct translators between the formats.

Recording Audit Trail

All of the systems looked at maintain some sort of record of the audit trail. In MDOOM the objects themselves store the history records. N-dim keeps a history of the revision of an object, and this may be viewed as a graph in the system. This feature is present in épée as well. Also a transaction log is maintained of the applied methods, which records the initial object, the method used and any resultant object.

2.3 The World Wide Web

The previous section examined several process engineering information management systems, against a list of requirements. This section discusses the World Wide Web (www), in particular examining data storage and user interaction.

The www was initially developed by Tim Berners-Lee [8] [10] at CERN (European Organisation for Nuclear Research) to access, in a single tool, information which was stored in various formats and different platforms within the organisation. It has now grown into a vast collection of interlinked documents spanning the globe.

The www and the Internet are two terms that are sometimes used almost interchangeably, but they are different. The Internet is formally defined by the Federal Networking Council [19] as, “the global information system that:

1. is logically linked together by a globally unique address space based on the Internet Protocol(IP) or its subsequent extensions/follow-ons.
2. is able to support communications using the Transmission Control Protocol/Internet

Protocol (TCP/IP) suite or its subsequent extensions/follow-ons, and/or other IP-compatible protocols.

3. provides, uses or makes accessible, either publicly or privately, high level services layered on the communications and related infrastructure described herein".

The www is simply one of the "high level services" layered on to the Internet. It is a client - server system.

The normal client for the www is a program called a browser. There are a variety of browsers available, of which the two most popular are Netscape Navigator [40] and Microsoft Internet Explorer [36]. The browser may send a request to any server on the www. Every file on the www has a Uniform Resource Locator (URL) [9]. This declares on what server the file is located, where in the server filesystem the file is, and what protocol should be used to retrieve it. For example the URL:

`http://www.chemeng.ed.ac.uk/people/ross.html`

is composed of the following:

Protocol `http` - hypertext transfer protocol.

Server `www.chemeng.ed.ac.uk` - the chemical engineering server at Edinburgh university, UK.

File Location `/people/ross.html` - the file `ross.html` in the directory `people` under the server document root.

Most text documents, or *pages*, on the www are written in HyperText Markup Language (HTML) [51]. HTML is a formatting language which controls the layout and display of text by means of tags included within the text, although not displayed, e.g. `<I>italics</I>` would be displayed by a browser as *italics*. Graphics files, sound and video clips, and animations may also be included in the HTML page. The URLs of other

files may be written in the HTML code using a tag called a hyperlink. Activating this hyperlink will then load the document referenced by that URL into the browser.

Any document may be put on the WWW by giving it a URL. However to be viewed in a standard browser, it must be one of a few particular types, e.g. HTML page, GIF image, etc. To view other files external viewers must be configured by means of a MIME (Multipurpose Internet Mail Extension) [22] type. In this situation all that is required is for the user to configure the browser such that if a file with a particular extension (e.g. .doc, .xls) is loaded into the browser, then either a given external program is launched and the file loaded into it, or the file is displayed using a specified plug-in to the browser.

XML

The eXtensible Markup Language (XML) [50] is a recommendation from the WC3 (the WWW standards committee [57]) for the next generation of web language, to supersede HTML. The biggest problem with HTML is that it is fixed. No new tags may be added without a revision of the standard, and hence a new version of HTML. Also it is very difficult to store information in HTML documents, in such a way that data may be easily extracted. XML is a means of storing information in the document. It does not concern itself with how the information stored is to be displayed. An extract from a simple document may look like the following:

```
<user>
<name>Ross</name>
<surname>Andrews</surname>
<email address=' 'ross@chemeng.ed.ac.uk' ' />
</user>
```

Each tag is defined in a Document Type Definition (DTD) file. This file contains a list of the expected tags, their format, and what data may be included between their opening

and closing tags. Each tag must either have a start and an end, e.g. `<name></name>`, or be defined as empty, e.g. `<email />`, in the DTD. Furthermore, every attribute value must be enclosed in quotes. Extracting data from a file in this format is relatively easy compared to the equivalent HTML file.

A distinction is made between an XML document being “well-formed” and “valid”. A well-formed document has a main start and end tag, and all other tags are either both opened and closed within this, or are empty. In addition no tags may be nested. A valid document also satisfies all of the above criteria. However it also must have a DTD and conform to it.

2.3.1 User Interaction

As well as displaying static pages, a browser can also support interaction with the user. There are two main methods of performing this interaction: either by server-side programming or client-side programming. The “server-side” and “client-side” refer to where the program enabling the interactivity will be executed, either on the server computer or the client computer. The program will use the processing power and memory of that computer.

Server-side Programming

One important HTML construct is the form. Upon “submission” it sends the data in the form elements from the browser to a server, which returns data to the browser. Forms are normally submitted by either pressing a button, or activating a link in the HTML page. A Common Gateway Interface (CGI) [38] program or script, which is run by the server, is required to process the form data. This script can run further programs on the local file system. This can be particularly useful for writing file data. However for security reasons, the server normally runs with restricted file privileges, i.e. can only write to, and read from, specific areas of the local file system. The operation is illustrated in figure 2.4. Forms allow some interactivity with the browser, in that

dynamic content can be written and returned to the browser by the CGI script.

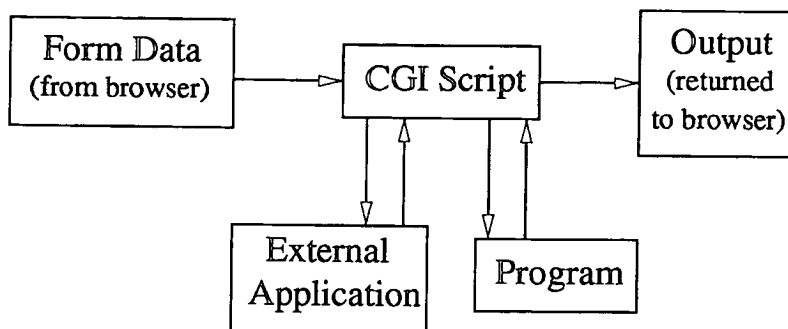


Figure 2.4: Processing Form Data

Client-side Programming

The other method of enabling interactivity is by using client-side programming. There are two main methods: using either Java applets or browser scripting languages.

Java Applets

Java [46] is a programming language, which is compiled into intermediate byte code. This code is then run by a Java virtual machine. This is intended to make Java code platform independent, without having to be recompiled by virtue of having Java virtual machines for different platforms. Applets are Java programs which are run by Java-enabled browsers. They have certain security restrictions, such as they cannot read from or write to the file system of the browser's computer. Applets may only read files from the server from which they were downloaded, and may only write files using CGI scripts. All Java programs should be able to be run by a Java virtual machine, regardless of the operating system on which the Java virtual machine is running. However with applets this does not seem to be the case. Certain applets will work in certain versions of browsers, and fail in others. This is clearly a drawback.

Browser Scripting Languages

There are a number of browser scripting languages, e.g. JavaScript [39], VBscript [37], but the only one that is supported by the majority of browsers is JavaScript. This is not to be confused with Java, which is a completely separate language. It can be used

for a number of things, such as manipulating form inputs, changing images in the page, or creating temporary new documents.

2.4 WWW-based Systems

Before developing a system which uses WWW technology, one must first examine what is possible using the WWW. All of the systems described in section 2.2.1 (MDOOM, n -dim, and *épée*) make at least some use of the WWW.

2.4.1 Use of the WWW in Process Engineering Information Management Systems

MDOOM

MDOOM has been integrated with the WWW in so much that the hyperviewer has been set up such that it may view both local and remote objects. Other than this, no use appears to have been made of the WWW.

N-dim

In the creation of n -dim, a lot of effort seems to have been expended. One has to ask is this duplicating work already achieved. Westerberg et al. [55] note that “the augmented second world (WWW) is the mindset behind the information modelling system we are creating called n -dim”.

Robertson et al. [42] make a comparison of n -dim with the WWW and note the deficiencies of the WWW, with regard to implementing n -dim. These were:

- controlling access to published data
- maintaining a history of object creation
- immutability of objects

This was the situation at the time the comparison was made. However, it is now possible to control access to pages, with a high degree of flexibility. But the other two deficiencies are still a major factor. Only a very basic history is kept, usually of pages visited. This is not organised into a tree structure, and is only maintained for a limited period. Also, there is no guarantee that any page on the WWW will remain at the same URL. It could be moved at any time at the whim of the file owner.

Westerberg et al. [55] discuss the interaction of n -dim with the WWW. Software has been written which identifies and saves WWW pages. Methods have been written which take the data from the WWW pages and construct n -dim objects from the information contained. Thus the interaction may be recorded, together with the intrinsic slot data, in n -dim.

Épée

Fraga and Laing [21] discuss the integration of épée with the WWW. A method is available which publishes an épée object as an HTML file. This may then be viewed in a WWW browser. If an épée MIME type has been set up, then the actual épée object may be viewed by activating the hyperlink to the épée object, and methods may be applied to it. However this is the only way to apply methods to the object. It is not possible to apply methods to the HTML version of the object.

2.4.2 Computer Supported Cooperative Work

The previous section showed how the various systems made use of the WWW. However, in all systems, the use of the WWW was at best just a feature of the system, rather than an integral part. One field which has aims similar to that of the system intended, maintaining the relationship between, and controlling access to, a linked set of documents, is that of Computer Supported Cooperative Work (CSCW). There are also systems developed for this field which use the WWW as an integral part of the system.

Computer Supported Cooperative Work has a number of requirements. These have

been listed by Dix [17] as:

- **Communication and control** - Cooperative work generally involves two or more people, accessing and possibly editing a common set of documents. It must be possible for users to communicate with each other, and also to control access to the documents such that only one person is editing a document at any one time, and all revisions are kept in a structured manner.
- **Meta-information: Understanding and Meaning** - In order for people to communicate, at the very least there must be a common vocabulary between them, such that no term is ambiguous.
- **Feedback and feedthrough** - In an interactive session, a user may edit a document. This will be stored as a revision of the original document. As well as storing the revisions, it should also be recorded that the original document has a revision, and the the revision came from the original document.
- **Awareness** - As well as being aware of who else is currently available for cooperative work, the system should also be aware of how the document was created/edited, by whom, and for what purpose.

These requirements are discussed below to see if they are met by standard WWW technology. If the requirements are not fully met, then WWW-based CSCW systems are examined to show how the WWW has been extended to meet these requirements.

Communication and Control

Users must be able to communicate with each other, and to be able to control access to documents such that two or more people are not simultaneously editing the same document. The WWW provides a communication protocol, and is ideal for accessing remote documents, indeed that was its original purpose. Access control may be enabled either by client location or password. However it is not possible to temporarily “lock” a single document, such that revisions may not be made to a document by two users at the same time. It is only possible to lock directories by adding specific information

on what clients to accept. Document locking has been implemented by a number of systems, e.g. BSCW [7], using server-side programming.

Meta-information: Understanding and Meaning

In order to cooperate effectively, users must have a common dialect, where terms are clearly understood by all parties. The www may be used to facilitate this in that ambiguous or new terms may be hyperlinked to a definition.

Feedback and Feedthrough

In cooperative sessions, it may require several revisions before a document will satisfy all participants. These revisions must be stored, and annotated that they are revisions of the original document, in case a user wants to go back to them. As well as seeing one's own revisions, known as feedback, the system should allow a user to see other people's revisions. This is known as feedthrough.

Maintaining links between documents is easily achieved using hyperlinks. However, they would have to be added to the documents, either manually or by the system. It is much more difficult to store the history of document access or creation.

Browsers have a basic history mechanism, in that the previous pages loaded are recorded for a specific time, see figure 2.5.

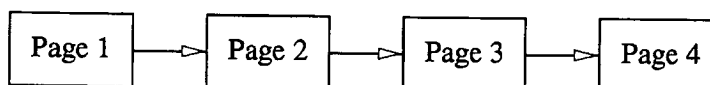


Figure 2.5: Browser History Illustration

The history is accessed by means of “back” and “forward” buttons on the browser. However if, for example, the user goes “back” from page 4 to page 2, and then loads page 5, the history branch from page 3 onwards is lost.

Version management has been implemented in BSCW [7], with each version being main-

tained unless explicitly destroyed.

Awareness

Any cooperative system must be aware of who else may cooperate on any given project. This can be something as simple as only permitting document access to a restricted group of people. As well as being aware of other users, the system should also be aware of what has happened, i.e. has a new document been created, or an existing one edited? Who did it, and why?

Restricting access to a limited group of people is relatively straightforward, either by password protection or client location. Recording the meaning of actions is much more difficult. Webmake [3] extends the HTML files to typed documents which contain extra information, such as most recent modification date. Yarn [56] uses an HTML logfile for similar information. BSCW has an event view for its workspace, which lists the contents of the workspace together with icons depicting the events that have occurred concerning the contents, e.g. read, edited, moved. Each document also has an information button which gives information on the document such as who created it, the date it was added to the system, etc.

2.4.3 CSCW Implementation on the WWW

In implementing CSCW on the www, Bentley et al. [7] list four main approaches:

1. **Purely www based** - These systems only use documented www technology, including Java, CGI, with no extensions to either the client or the server, e.g. WebMake [3].
2. **Customised servers** - Systems in this category extend the functionality of the server, but may still be connected to by standard clients, e.g. Trevor et al. [49] describe a system following this approach.
3. **Customised clients** - Add-ons are written for the client to extend its function-

ality. The server may also be customised, e.g. Oblets [12].

4. **www-related** - These use the WWW as a user interface, but little else, e.g. PageSpace [14].

The flexibility of the system developed increases from one to four. Having a purely WWW based system means that all the advantages of the WWW are present. However, the system is limited by the fact that it must work using both standard browsers and servers. Customising the server introduces some flexibility into the system. It does mean that the system will only work on one particular server. However this is not that great a disadvantage in that any standard client may still connect to the system. Customising the client leaves the system a lot more flexible in terms of presentation and functionality. However add-ons have to be written for each of the desired clients, on each of the major operating systems. Also, installing these add-ons may not be a trivial task. The greatest flexibility is offered to the designer by WWW-related systems, in that the communication protocols and the graphic interface may be customised, but also the greatest amount of work!

2.5 Discussion

All of the systems described in section 2.2 have duplicated at least some of the functionality of the WWW. The question arises as to how many of the process engineering information management system requirements are fulfilled by the WWW as it stands, and could any of the remaining criteria be met with a little extra effort in developing a new system.

In basing any proposed system around the WWW some of the criteria for a process engineering information management system have been met by default, others have been partially met, and finally some criteria have not been met at all.

Distributed System

The www is a client-server system which is by its very nature distributed. The computing capabilities of both the client and the server may be utilised in any system, and providing that the system is purely www-based, it may be accessed from anywhere on the www.

Simultaneous Platform Support

Due to the popularity of the www there will almost certainly be a browser available for all but the most obscure operating system. This removes one of the more tedious aspects of developing new systems, that of providing a user interface.

Efficiency

There are three facets to this criterion: access time, time to run simple tasks and the time taken to learn the system, including the interface. Most people are familiar with www browsers and are comfortable with the aspects of www pages, e.g. hyperlinks, etc. Thus the interface should be intuitive.

However, using the www may mean slow access times to the system. This is caused by network congestion, and depends both on the volume of network traffic to and from the site, and the size of the network connection. Having a bigger network connection may alleviate some of this, but it will not help a remote user with local network congestion. One method of dealing with this would be to reduce the required network traffic.

The time taken to run simple tasks depends on the computer upon which the tasks are being run, either the server or client. Conceivably if too many clients attempted to run programs on the server, this could significantly affect running times. However, with a relatively powerful server, the time taken to run server-side programs is usually minor compared to the time taken to access the server.

Persistent Data Storage

This requirement is not met simply by using the WWW, and will have to be dealt with by any system developed. All of the systems looked at in this chapter use objects to encapsulate the data and then store the objects in databases.

Are there any significant advantages in storing the data in databases? One major disadvantage is that database files cannot be directly viewed on the WWW by browsers. They may be viewed by standard browsers if the data is transformed into HTML pages by the server. However this increases the computing load on the server. In effect a viewer has to be written to access the data. Also the user would not be able to extract files and view them on their own systems without having at least some database technology installed. Reasons for using databases would be to manage the data if there are a large number of data sets, or if the data needs to be searched. Initially, it is not intended to have a large number of data sets, or to provide a search facility. Thus there seems to be no advantage to using databases.

Objects provide a convenient framework for encapsulating the data in familiar structures, e.g. components, streams, etc. The WWW however consists of pages which are simple HTML files. The question arises could the idea of objects and files be combined? If the concept of objects could be maintained, yet stored using a simple file accessible by any browser, then the best of both worlds would be achieved.

User Extensible

In the WWW as a whole, anyone can add pages. Whether a system based on the WWW is extensible or not depends on the system. For a new system, this would mean both adding new data structures, and extending existing structures.

Application Integration

Basing a system around the WWW does not automatically fulfill this requirement. The most convenient method of achieving application integration would be using the second method discussed in section 2.2.6, i.e. that of encapsulation of tools. This would involve writing translators between the storage format and the tool format, and vice versa.

Record Audit Trail

Most browsers make at least some record of which pages have been visited recently, and it is possible to move backwards and forwards using this list. However it is not possible to generate a history tree, i.e. record more than one path. Also this history is only maintained for a specified time.

Several WWW-based systems have managed to record some sort of audit trail, or at least to record enough information such that one may be generated. Webmake [3] include extra information in the page, such as the date the page was last modified. This is similar to the intrinsic slots in n -dim. Yarn [56] stores similar information in a logfile. Thus it is possible to maintain a framework where this information is recorded.

2.6 Summary

It has been shown that using WWW technology to develop and operate a process engineering information management system fulfils several requirements of such a system by default. The system itself must fulfill those requirements that have not been met, or only partially met. These are: to be efficient, to have persistent data storage, to be user extensible, to facilitate application integration and finally to record the audit trail. A system which attempts to fulfill these criteria is described in the next two chapters.

Chapter 3

Design of a WWW Based Model Building System

3.1 Introduction

The previous chapter discussed process engineering information management systems. It showed how many of these systems use objects to provide a convenient mechanism to encapsulate process engineering data. This chapter discusses the design and implementation of a system intended to chronicle process engineering model building using www technology.

A set of standard objects representing process engineering entities were developed. These objects are HTML files, and are intended to represent entities such as components, units, etc.

Specific objects are discussed in detail in the next chapter. This chapter first describes the structure of generic objects. Next object creation and manipulation is discussed. The mechanics behind entering the system is detailed. Also detailed is a typical method application. The chapter concludes by describing sections of objects which are common to all object classes.

3.2 Object Structure

An object consists of three main sections: data slots, logistical slots and methods. A distinction is made between the data slots and the logistical slots. The data slots contain the data for the object, whilst the logistical slots contain data on the data e.g. who created the object, at what time, etc., cf. intrinsic slots in n -dim.

3.2.1 Data Slots

Each data slot has a number of attributes. For a simple data slot these are:

- **Name** - The name of the slot.
- **Value** - The slot value.
- **Description** - A text description of the slot, normally the units of the slot value.
- **Status** - This attribute shows how much trust may be put into a slot.

The status attribute may have one of several values: *unassigned*, *entered by user*, *estimated*, *obtained from databank*, *incomplete object* or *complete object*.

Initially, when a slot is created, it has status *unassigned*. The user may enter data to the slot, which after processing changes the status to be *entered by user*. This means that the slot data may not be altered. Similarly the slot may not be edited if the status is *obtained from databank*. However, if the data in the slot has been *estimated*, then the data will still be displayed to the user. However it may be edited by the user. Upon editing its status changes to be *entered by user*.

The remaining two status values refer to slots which contain objects. If the object has all of its constituent slots filled, ie. there are no slots with status *unassigned*, then the status of the contained object is *complete object*. Otherwise the status is *incomplete object*.

Name:	Acetic_acid	Status: entered by user
Abs_formula:	C2H4O2	Status: obtained from databank
CARN:	[000064-19-7]	Status: obtained from databank
Aspen_name:	ACETIC-ACID	Status: obtained from databank
Struct_formula:	CH3COOH	Status: entered by user
Molec_weight:	60.05000	Status: obtained from databank
Boiling_pt:	391.0500	Status: obtained from databank
Cp_a:		Status: unassigned
Cp_b:		Status: unassigned
Cp_c:		Status: unassigned
Cp_d:		Status: unassigned
Crit_temp:	587.25 K	Status: estimated
Crit_pres:	57.31 bar	Status: estimated
Crit_vol:	71.50 cm ³ /mol	Status: estimated
Gibbs_fe:	-377.95 kJ/kmol	Status: estimated
Enthalpy_form:	-434.88 kJ/kmol	Status: estimated

Figure 3.1: The data slots of a partially complete component object

To make the difference in the slot status values clearer, each slot is displayed in a colour dependent upon its status. Slots with status *unassigned* or *incomplete object* are red, whilst *estimated* slots are coloured green. All other slots are displayed in the default colour, normally black. This is illustrated in figure 3.1.

All objects are immutable. Once created, the data in an object may not be modified. Instead, if the user wishes to edit the object data, the data are copied, edited and a new object created with the altered data. The new object is said to be a child of the original object, the parent object.

3.2.2 Logistical Slots

The logistical slots hold information on the information. This may be considered analogous to the slots in both *épée* and *n*-dim. The logistical slots consist of:

- **Object Creator** - This is the email address of the user, which provides a unique global identifier for the user.

- **Method Used** - This slot contains the name of the method used to create the object. If this is a new object, then this is indicated here.
- **Parent Object** - If this is a new object, then there will be no parent object. Otherwise the object will have been created by modifying the data in another object. This slot contains a hyperlink to that object.
- **Children** - If the data in this object have been modified, then a new object will have been created. This slot contains hyperlinks to those objects, if any.
- **Date Created** - The time and date at which the object was created is stored in this slot.
- **Object Name** - This slot contains the name of the object. How an object name is selected is discussed in section 3.3.3.

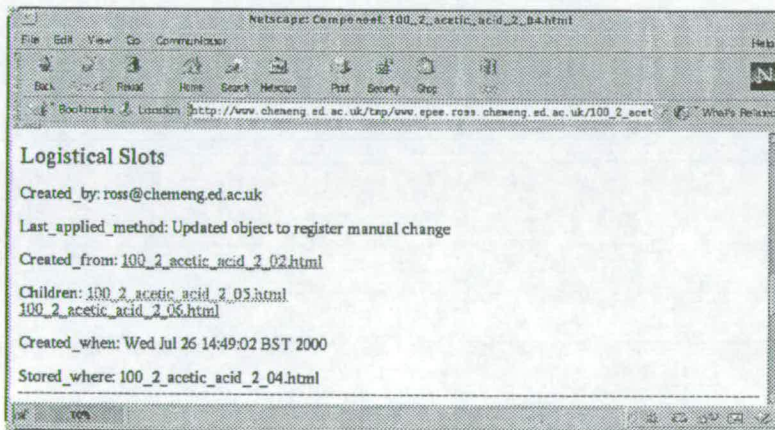


Figure 3.2: The logistical slots of an object

3.2.3 Methods

The methods section is divided up into three separate parts:

- **Object Methods** These methods manipulate the object data. Each class of object has a different set of object methods.
- **Other Selection Options** These options do not use the object data, and so are not strictly methods. They are common to all classes of the standard objects.
- **History Options** These options allow the history of object creation and manipulation to be viewed in a variety of ways.

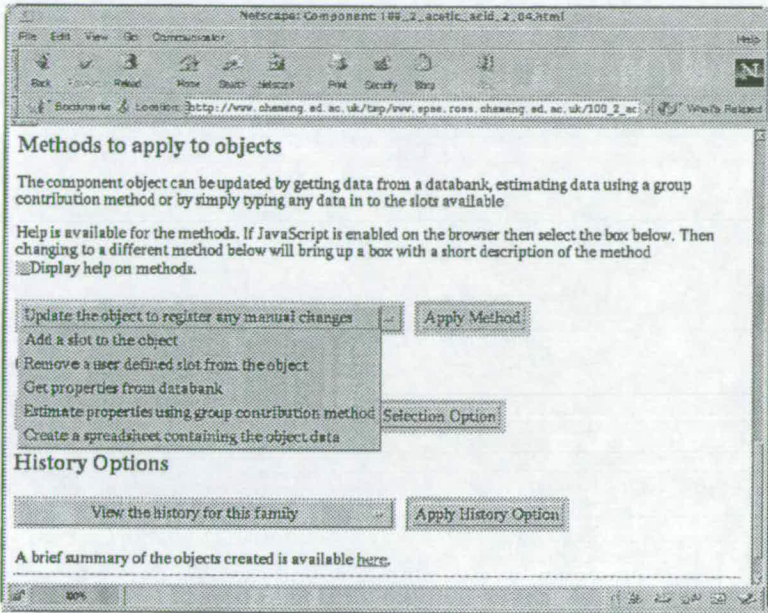


Figure 3.3: The methods of a component object

A brief description of each method is available online by checking the help button, and selecting the method for which a description is required (see figure 3.4).

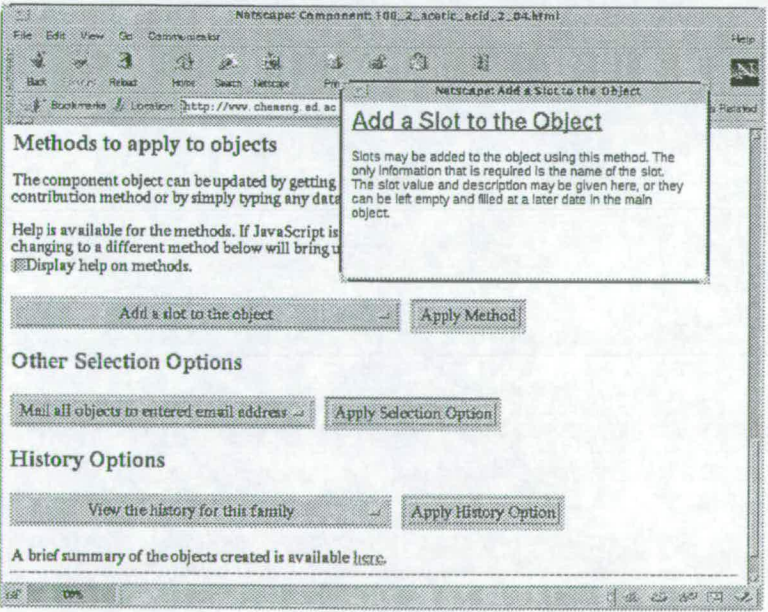


Figure 3.4: The help function for object methods

Each of the object methods will be discussed in the next chapter. Selection options and history options are described in sections 3.4 and 3.5 respectively.

Objects are stored as HTML files. This is discussed in detail in appendix D. Briefly, form elements are used to hold the data within HTML files, with each slot attribute requiring a separate form element. If a data slot contains another object, then this slot is linked rather than included in the object. This cuts down on the storage space required.

The decision to use a file-based approach was made so that to view the files no manipulation or data extraction is required. The file may be loaded into any web browser and immediately displayed. However to access the data programmatically this file has to be parsed and the data extracted, which will slow down method execution. This time may be significant if a large number of object files need to be parsed.

There are also certain occasions where it is necessary to store object data in a file during method execution, e.g. if requesting information from the user. Rather than write an HTML file and have to parse it again, an XML file is written which is easier to parse.

3.3 Object Creation and Manipulation

The previous section described the structure of a generic object. This section will examine the creation and manipulation of the same generic objects. There are two ways of creating an object: either by using the model generation manager, or by applying a method to an existing object. Both of these techniques are detailed here.

The model generation manager is initially displayed to the user when they enter the system. Thus the first part of this section describes the options available to the user when they enter the system, and also what happens behind the scenes.

3.3.1 Entering the System

Upon entering the system the user is prompted for their email address. This serves two functions:

1. It provides a unique global identifier for the user.
2. It enables any created objects to be sent to the user at a later date.

This is a session organised system. When the user enters the system, a new session is started for that user. At this point an identifier is generated for the session. This comprises of two numbers, separated by an underscore ('_'), and prefixes the names of all objects created in this session. The first number is the number of years since 1900, and the second number is the number of sessions that have taken place in the current year, e.g. 100_234. This second number is incremented after the start of each session.

The system uses the email address supplied to generate a directory name, which will be use to create a temporary directory on the WWW server which serves as the user's workspace. This directory will be deleted after a twenty-four hour period where no objects are created. Upon entering the system, if the directory still exists, then history trees (see section 3.5) are generated for all of the objects in the workspace and displayed to the user.

If the directory does not exist, then it is created, and the user is prompted to supply a password to restrict access to the directory. If a password is given, then the user name will be the email address supplied when entering the system. Supplying a password is optional. If no password is given, then anyone may access the created objects without restriction.

Next several component objects, and two mixture objects are created to provide a starting point for the user. Then the user is presented with a choice. They can either go directly to the model generation manager, where models are entered and generated, or there is a more gradual introduction for less experienced users. The user is led

through the creation of several objects of different classes, before the model generation manager is displayed.

This procedure is illustrated graphically in figure 3.5.

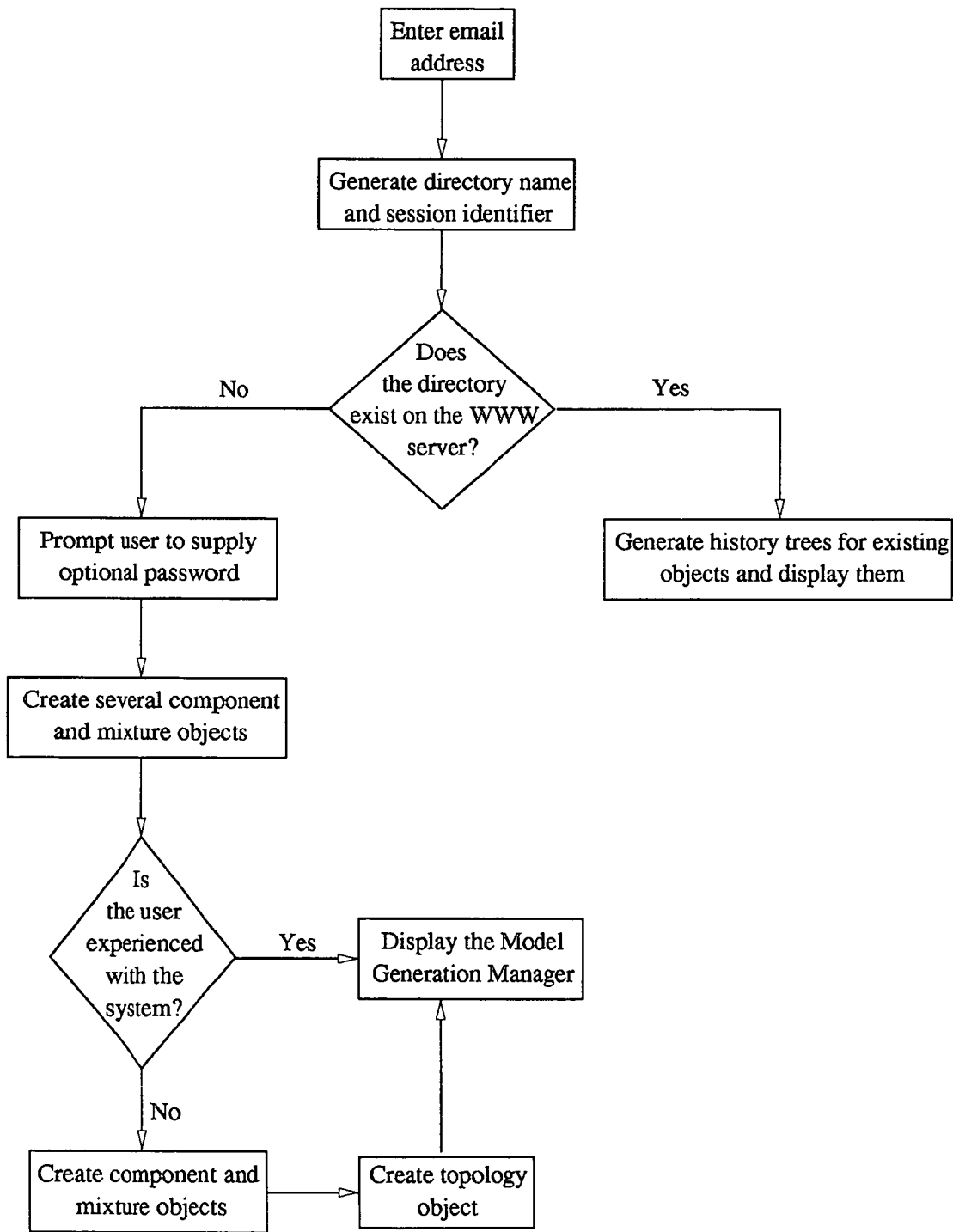


Figure 3.5: Starting a Session

Model Generation Manager

The model generation manager enables the user to create models from existing objects, and also to create objects without applying methods to an existing object. It consists of four sections: physical property input, data input, model creation, and manual object creation.

The physical property section enables the creation of objects representing standard physical property entities: components and mixtures. The data input section allows the creation of topology objects which contain the data for a process. Once topology objects have been defined, simulation models of these processes may be created in the model generation section. The final section, the manual object creation section, is where individual objects of any class may be created.

Physical Property Input

This section allows the user to create mixture objects, by selecting components from a list (see figure 3.6). A basic databank contains some data for these components: boiling point, molecular weight, ASPEN name, CARN (Chemical Abstracts Reference Number) and molecular formula. If any components are not on this list, they may still be entered here, but none of the above data will be available.

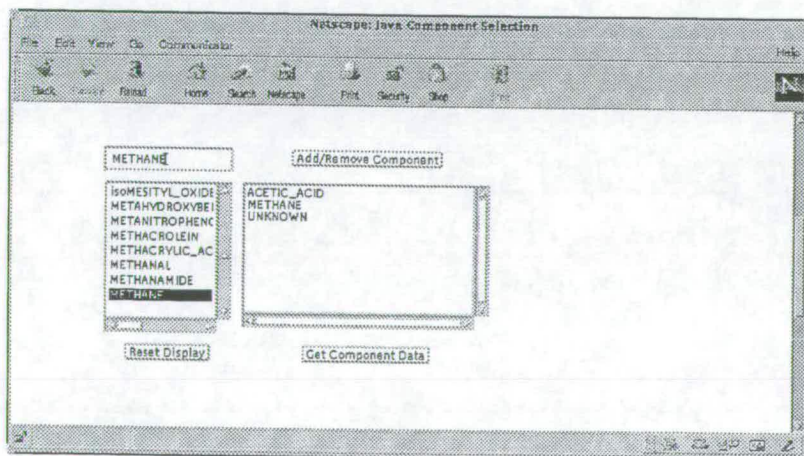


Figure 3.6: Selecting component names

Once the components have been selected, the user is then prompted to enter the structural formula of each of the components. The structural formula may either be entered by hand, or built up using a “molecular typewriter” (see figure 4.2.5). This is discussed in detail in section 4.2.5.

After the properties have been estimated, the components are combined to create a mixture object.

Data Input

The second section of the model generation manager is the data input section. The data input area is where models may be entered to create a topology object, and associated stream, and unit objects. This is accomplished in one of two ways: either creating a new topology object, or by parsing an ASPEN file.

To create a new topology object, the user must first of all select a mixture object. The components in this mixture object will be used to create the necessary stream objects of the topology. Next the user can select an approximate structure which can then be modified as needed (see figure 3.7). These approximate structures correspond to a surprisingly large amount of processes.

The transforming of an approximate topology into a topology object, together with the required stream and unit objects, is discussed in section 4.7.1.

Alternatively no structure may be specified, and the topology object created with no structure. This may be entered at a later date.

The second method, that of parsing an ASPEN file, involves taking one of a number of formats of ASPEN files (backup file, input file or summary file). This is then parsed, and information extracted to create component, stream, and unit objects, together with the topology object. A proper discussion requires some detailed knowledge of the standard objects, which will be left until section 4.7.13.

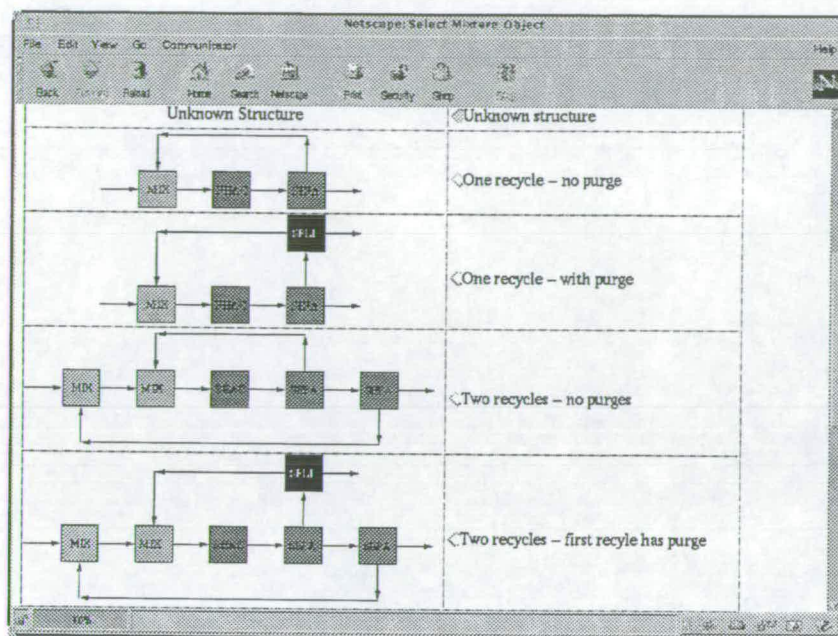


Figure 3.7: Selecting an approximate structure

Model Creation

This section allows the user to create models from any topology object in the user's workspace. At least one topology object must be created before any models may be created. The model types that may be created are: ASPEN model, a HYSYS model, a spreadsheet model and an equation model. Apart from the equation model, these model types may also be created by applying methods to a topology object. They are discussed in detail in sections 4.7.11, 4.7.12 and 4.7.9. The equation model is an actual object, and its creation is described in section 4.8.

Manual Object Creation

The final section of the model generation manager allows the user to manually create individual objects of any class. There are certain restrictions on the order that objects may be created, e.g. mixture objects may not be created until component objects have first been created. This section is primarily intended for the creation of objects of classes specified by the user.

3.3.2 Method Application

The other way of creating objects, apart from the model generation manager, is by applying methods to existing objects. The methods for each of the standard objects are discussed in chapter 4. This section will discuss the mechanics of a typical method application, i.e. what happens from when the method is initially applied until the new object appears in the browser window.

All methods are applied by submitting a form to the WWW server. This is processed by a CGI script. When the CGI script is activated, it initially checks to see if the object to which the method is being applied still exists in the user's server workspace. This might not be the case if the user saved objects created in an earlier session, which have subsequently been deleted from the user's workspace. The user could then have loaded the saved object in a browser and attempted to apply a method to it.

If the object does not exist in the user's workspace, then a check is made to see if the user has started a new session, i.e. has a directory on the WWW server. If not, then the user is prompted to enter their email address and start a new session. A new session is not automatically started as the user would not have the option to give a password to restrict access to objects created in this session. If a directory does exist, then a check is made to see if the object has a local URL, e.g. `file:/home/ross/objects/.....`, or a full URL, e.g. `http://www.chemeng.ed.ac.uk/.....`. If the object has a full URL, then the object is uploaded to the user's server workspace, together with any objects referenced by that object (excluding parent and children objects). This uploading of referenced objects is applied recursively until all objects referenced by any of the uploaded objects have been uploaded. This is discussed further in section 3.4.4. If the original object has a local URL, then it may not be uploaded, and a message is displayed to the user informing them of this.

Once the object has been located in the user's workspace, it is parsed to extract the object data. These data are compared to the data submitted in the form to check for any changes made by the user. If there are any differences in the two data sets then

it is assumed that the user has made changes to the object data, and the object is updated accordingly.

The next task is to extract the data necessary to apply the method from the object slots. Slots accept any data. For some methods, data of a particular type is required in a slot, e.g. integers, a component object. If data of an incorrect type is present in the slot and detected by the method, then an error message is generated and displayed to the user. It is the responsibility of the individual method to check for required data, and data types, and to generate the appropriate error message. The error message is displayed in the form of an HTML page displaying the message, together with a link to the object to which the method was originally applied.

After all this, the actual method may finally be applied. This could involve running another program, an external application, or something as simple as a subroutine. Any messages generated by the method code are trapped, and if the method has not been applied successfully, then these messages are displayed to the user.

If the method has been successfully applied, then the new object is created, and registered as a child of the original object. Finally the new object is displayed in the browser.

This procedure is illustrated graphically in figure 3.8.

3.3.3 Object Naming

Each object is given a name which is intended to be both globally unique, and to provide some information on the object. It consists of four numbers and one character string together with the .html file extension, e.g. 101.23_stream1.01.html. The first two numbers represent the session identifier (see section 3.3.1). Next comes the character string. For most objects, this will be the object class. However for component objects, the string is the name of the component. Any characters which are not letters or numbers (e.g. spaces, commas, hyphens, etc.) are replaced by the underscore

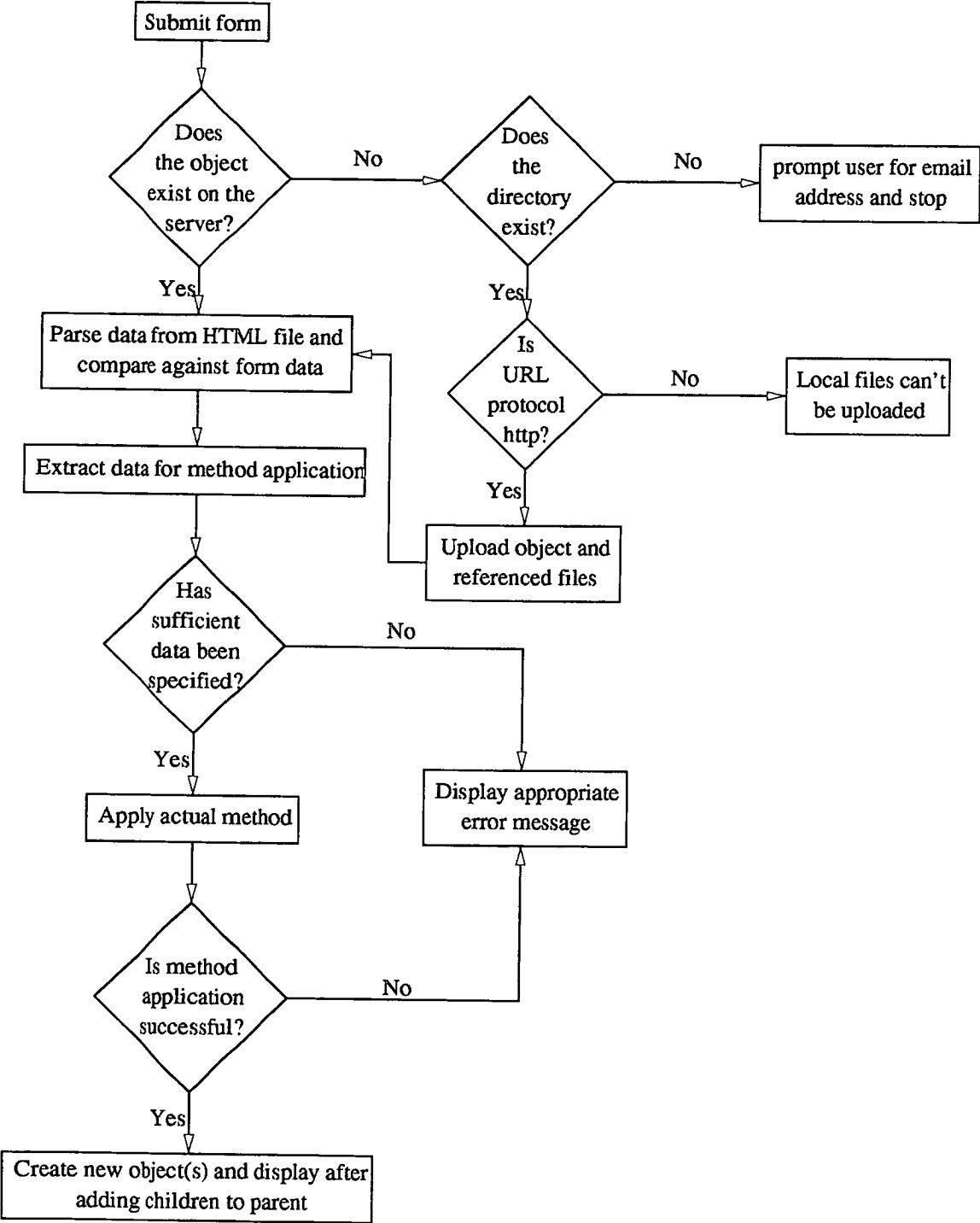


Figure 3.8: Applying a method to an object.

character. Following the string, there is another number. This is the family number, and represents the number of original objects, i.e. those that have no parent, created in that class (or component for the component class) of object. This is incremented each

time a new original object is created. These three numbers and the string comprise the family name of the object, e.g. 101_23_stream1. The final number represents the number of objects created in the particular family. This number is incremented after each new family object is created.

When an original object is created, it is given a new family name, and a child number of 01. Any objects created from this object in this session, either directly, or via several intermediate objects, will have the same family name, but a different child number. The child number will be incremented until an unused number is obtained.

3.4 Other Selection Options

This section discusses the other selection options. These are not strictly methods, as they do not use the object data. However they are useful for managing objects. Each of the selection options are discussed in turn below.

3.4.1 Mail all objects to entered email address

This selection option enables the user to have all the objects stored in the user's workspace sent to the entered email address. Before being emailed, the objects are compressed into a single zip file. As sending large files via email may cause problems, the user is informed of the file size, and given the option of either having the zip file emailed, or downloaded from a hyperlink.

3.4.2 Create a new object

Selecting this option displays the model generation manager (see section 3.3.1) to the user, where new original objects may be created.

3.4.3 Finish the session

The user may end the session by selecting this option. This removes all of the objects stored in the user's workspace. Before doing so, a confirmation request is displayed to the user to confirm that the objects should be deleted.

3.4.4 Obtain an object from a URL

If the user saves objects by having them sent to the entered email address, they may be published on the WWW at a later date. However to apply a method to an object, it must exist in the user's workspace. Selecting this option displays a text box to the user. If the URL of an object is entered into this textbox, then it will be uploaded to the user's workspace on the server. Also uploaded will be any objects, other than parent or child objects, referenced by this object. It is assumed that these referenced objects are in the same directory as the original uploaded object. This is carried out recursively on the referenced objects, until all objects referenced by any of the uploaded objects have been uploaded. The URLs of any parent and child object are changed to point to the location from which the objects were uploaded. Note that when history trees for families of uploaded objects are generated, it is assumed that the uploaded object is an original object, i.e. it has no parent.

The recursive downloading of web links is carried out so that all objects linked from the original object are present in the user's workspace. Thus if any data from these objects are required for subsequent method applications, the object may be parsed locally. This speeds up and simplifies these method applications, as it means that objects only exist in one location (the user's workspace) and not scattered throughout the Internet. However this approach does slow down the uploading of the initial object.

3.4.5 Create a new custom class

Section 3.2 has described the structure of objects. To be a true object-oriented system, the objects must support inheritance. This is described here. A base object has been defined, from which all the other standard objects inherit both slots and methods. This is illustrated in figure 3.9.

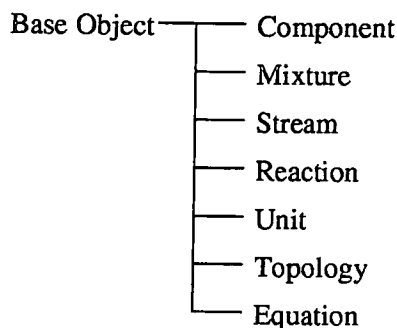


Figure 3.9: Object Inheritance

The class data are stored internally in an ASCII file. This file contains the names of the class methods, together with the slots and the slot attributes. It also lists the object class, if any, from which the current object class inherits. A description of the structure of class files is contained in appendix C. This appendix also contains the class files of the standard objects.

The methods of a standard object are processed by a single CGI script for each object class. The class CGI script is passed a parameter string which indicates the appropriate code to execute. The inherited methods are included as a library of subroutines. By altering the parameter string of an inherited method to point to different code, it is possible for an inherited method to have its behaviour redefined in the class CGI script. This is known as polymorphism, where derived classes may redefine inherited methods.

This selection option provides a forms-based interface to create a new class file. The user is first asked to provide the numbers of the slots and methods that are to be defined in the new object class. Once this has been confirmed, the system generates the required number of form elements to input the desired new slots and methods. Only two elements of the new class are compulsory: the class name, and the object

class from which this class inherits. If the new class does not inherit from any of the standard object classes, then it must inherit from the base object class.

The methods of a standard object are processed by a single CGI script for all of the methods. However, whenever a custom method is defined, it must be processed using its own CGI script. This CGI script must be supplied by the user, and executed on a server on which they may run CGI scripts.

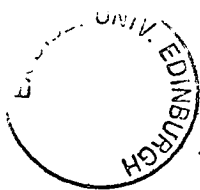
The main reason for implementing custom methods using this procedure is that all methods consist of executable code. If this code were to be run on the local chemical engineering server, a number of problems and security issues would arise. If the method uses a compiled program, it would have to be compiled for the correct platform. If the method is an interpreted script, then the interpreter would have to be running on the local server. The security issues relate to the fact that a piece of code of unknown origin running on the server could cause malicious, or even accidental, damage. Making the user run the code on their own server removes these problems and limits the security risk.

Applying the custom method results in the object data being written in XML format (see appendix D). Any objects linked, either directly or via an intermediate object, from this object are also written in XML format.

Two parameters are passed to the user-supplied CGI script:

- `xml_file` - the URL of a file containing the object data in XML format.
- `url_path` - the URL of the directory where the user's objects are located.

The user's CGI script can then upload the XML file and apply the custom method to alter the data. The CGI script should return the URL of the altered XML file, in a single line, and in plain text. This XML file will be uploaded to the user's workspace on the server, the data parsed, and the new object created. Any objects referenced in this new object are dealt with as follows:



1. If the object exists in the user's workspace then it is simply referenced.
2. If it exists in the same directory from where the modified XML file was uploaded, it will be uploaded to the user's workspace.
3. Finally, if no object is found, the same directory from where the original XML was uploaded is searched again. This time for a file of the same name as the object, but with an `xml` extension, rather than an `html` extension. This file is uploaded, the data parsed and a new object created.

These steps are carried out recursively on any XML file uploaded to the user's workspace. This is illustrated in figure 3.10

3.5 History Options

The history options are:

- View the history for this family
- Show the history for all class objects
- Show the history of all created objects
- Show logistical information for this class of object
- Show logistical information for all classes of object

These options enable the history of object creation to be viewed in two different formats. Firstly tree structures may be viewed for the object family, for the families of all objects in the class, and for all families of all classes, see figure 3.11.

All objects are displayed as hyperlinks. Activating the hyperlink loads the selected object into a new browser window.

The other format for displaying the history are the manifest lists. These display in a tabular format, arranged chronologically, the data in the logistical slots for either

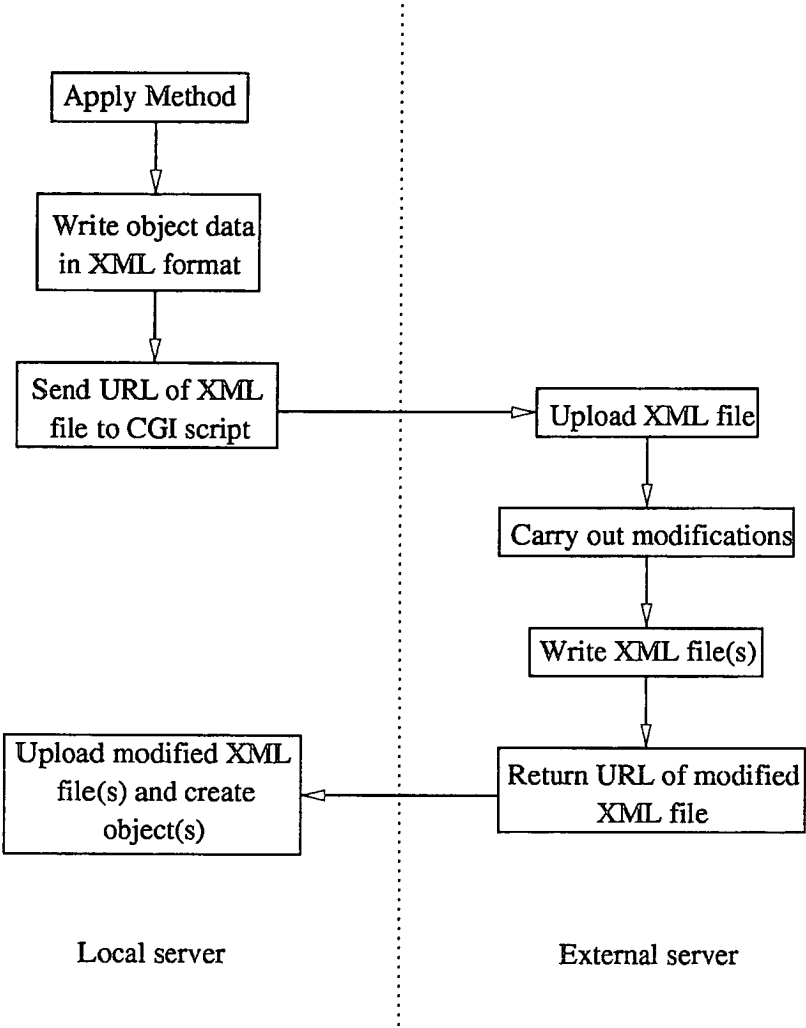


Figure 3.10: Applying a custom method

a class of object, or for all classes of objects. For sets of large objects, it will take a significant time to extract the data from the relevant objects and display it. In order to display some data, the data are displayed in tables, with each table holding no more than fifteen objects, see figure 3.12.

Both of these arrangements, the history trees and the manifest tables, are only written if the user specifically requests to view them. However there is a log file in which a record is maintained of object creation. This file may be viewed by the user, and contains a summary of all the objects created in the user's workspace up to that point. Data is inserted to this file every time an object is created, and shows the object name, the method used to create the object, and the time and date that the object was created

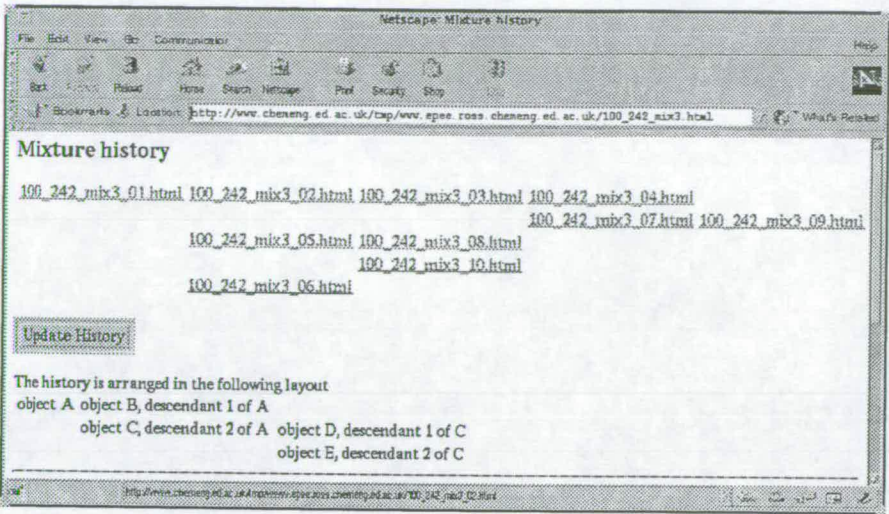


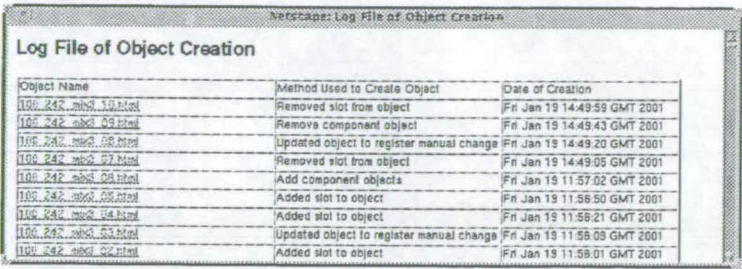
Figure 3.11: A history tree for an object family

The screenshot shows a Netscape browser window titled "Netscape: Manifest for Mixture Objects". The address bar shows the URL "http://www.chemeng.ed.ac.uk/tmp/www.spee.ross.chemeng.ed.ac.uk/100_242_mix3.manifest.html". The main content area is titled "Manifest for Mixture Objects" and "Manifest for Mixture Objects (1/16)". Below the title is a table with 6 columns: Name, Created by, Last Applied Method, Parent, Children, and Date Created. The table contains 10 rows of data.

Name	Created by	Last Applied Method	Parent	Children	Date Created
100_242_mix3_01.html	ross@chemeng.ed.ac.uk	new object	new object	100_242_mix3_02.html 100_242_mix3_05.html 100_242_mix3_06.html	Fri Jan 19 11:54:37 GMT 2001
100_242_mix3_02.html	ross@chemeng.ed.ac.uk	Added slot to object	100_242_mix3_01.html	100_242_mix3_03.html	Fri Jan 19 11:56:01 GMT 2001
100_242_mix3_03.html	ross@chemeng.ed.ac.uk	Updated object to register manual change	100_242_mix3_02.html	100_242_mix3_04.html 100_242_mix3_07.html	Fri Jan 19 11:56:09 GMT 2001
100_242_mix3_04.html	ross@chemeng.ed.ac.uk	Added slot to object	100_242_mix3_03.html	none created	Fri Jan 19 11:56:21 GMT 2001
100_242_mix3_05.html	ross@chemeng.ed.ac.uk	Added slot to object	100_242_mix3_01.html	100_242_mix3_08.html 100_242_mix3_10.html	Fri Jan 19 11:56:50 GMT 2001
100_242_mix3_06.html	ross@chemeng.ed.ac.uk	Add component objects	100_242_mix3_01.html	none created	Fri Jan 19 11:57:02 GMT 2001
100_242_mix3_07.html	ross@chemeng.ed.ac.uk	Removed slot from object	100_242_mix3_03.html	100_242_mix3_09.html	Fri Jan 19 14:49:05 GMT 2001
100_242_mix3_08.html	ross@chemeng.ed.ac.uk	Updated object to register manual change	100_242_mix3_05.html	none created	Fri Jan 19 14:49:20 GMT 2001
100_242_mix3_09.html	ross@chemeng.ed.ac.uk	Remove component object	100_242_mix3_07.html	none created	Fri Jan 19 14:49:43 GMT 2001

Figure 3.12: A manifest table for mixture objects

(see figure 3.13).



Object Name	Method Used to Create Object	Date of Creation
100_242_mbs_10.htm	Removed slot from object	Fri Jan 19 14:49:59 GMT 2001
100_242_mbs_09.htm	Remove component object	Fri Jan 19 14:49:43 GMT 2001
100_242_mbs_08.htm	Updated object to register manual change	Fri Jan 19 14:49:20 GMT 2001
100_242_mbs_07.htm	Removed slot from object	Fri Jan 19 14:49:05 GMT 2001
100_242_mbs_06.htm	Add component objects	Fri Jan 19 11:57:02 GMT 2001
100_242_mbs_05.htm	Added slot to object	Fri Jan 19 11:56:50 GMT 2001
100_242_mbs_04.htm	Added slot to object	Fri Jan 19 11:56:21 GMT 2001
100_242_mbs_03.htm	Updated object to register manual change	Fri Jan 19 11:55:09 GMT 2001
100_242_mbs_02.htm	Added slot to object	Fri Jan 19 11:55:01 GMT 2001

Figure 3.13: The logfile of object creation

3.6 Summary

This chapter has discussed the instantiation and management of objects using an object-oriented system accessed using WWW technology. It has described the structure and methods of a generic object. It has also demonstrated how the concept of an object may be combined with HTML to provide a structure capable of storing the required data.

The next chapter will discuss in detail the slots and methods of a set of objects representing process engineering entities.

Chapter 4

Process Engineering Objects

4.1 Introduction

The previous chapter described the structure of a generic object, and how objects were created and manipulated. This chapter describes the data slots and methods of the set of standard process engineering objects. These are:

- component
- mixture
- stream
- reaction
- unit
- topology
- equation

The names of the object set, together with the individual object data slots are derived from the set of object definitions developed by the creators of *épée* [15]. The object set used here is a much simplified version, both in terms of object classes and their data slots, to reflect their proposed purpose.

These objects are primarily intended for use in the initial stages of a design, where it may be necessary to evaluate several alternative designs, or part designs, in a short period of time. To provide approximate information for these designs, short-cut non-rigorous procedures may be used to calculate data relatively quickly. These are provided in the object methods and data slots. As design proceeds to more detailed stages, more rigorous methods would be required to calculate accurate data, together with a greater variety of objects to encapsulate and manipulate the resultant data.

The data slots and individual methods of each of these objects are detailed in turn.

4.2 Component

The component object is intended to store the data of an individual component. It contains slots for the following component data:

- Name
- Absolute formula
- CARN - the Chemical Abstracts Reference Number
- ASPEN name - the name of the component as stored in the ASPEN databank
- Structural formula - the component formula arranged in structural groups, e.g. CH3COOH for acetic acid
- Molecular weight
- Atmospheric boiling point
- Ideal vapour heat capacity polynomial terms - used in the formula $a + bT + cT^2 + dT^3$ to calculate the heat capacity at a given temperature, T
- Critical temperature
- Critical pressure
- Critical volume
- Gibbs free energy of formation
- Enthalpy of formation

The methods for component objects are:

- Update the object to register any manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Get properties from the databank
- Estimate properties using group contribution method
- Create a spreadsheet containing the object data

4.2.1 Update the object to register any manual changes

This method processes the object slot data to take into account any changes that have been made. The status of any altered slots are changed to be *entered by user*. If no changes have been made to the slot data then the user is informed, and an error message displayed. Whenever a method is applied to an object, the data slots are checked and updated, before the selected method is applied to the updated data.

4.2.2 Add a slot to the object

The user may add to the list of built-in data slots by applying this method. Upon applying the method the user is presented with boxes to enter the slot name, value and description, see figure 4.1. The slot name is required, otherwise an error message is displayed. Optionally, the slot value and a description may be entered at this point. If no value is given, then the slot is added with status *unassigned*, and any given description. A value and description may be entered at a later date. If a value is given, then the description must either be entered here or not at all. Once the slot is added with a value the slot status becomes *entered by user* and no further change may be made to the slot. Only one slot may be added at each method application.

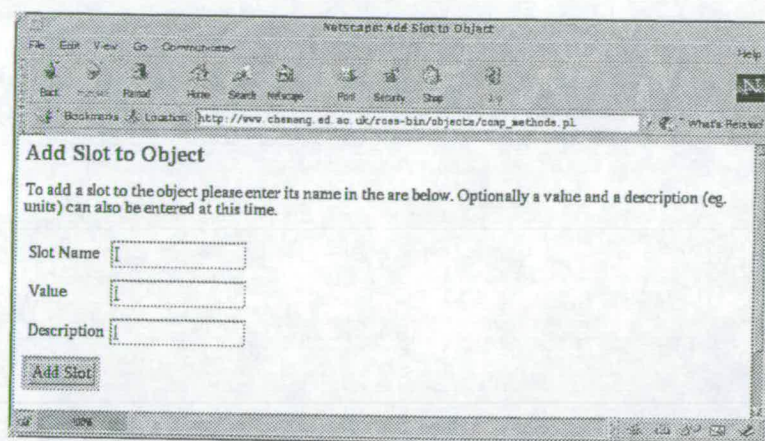


Figure 4.1: Adding a slot to an object

4.2.3 Remove a user defined slot from the object

As well as adding slots to the object, the user may also remove any slots which they have defined. No built-in slots may be removed using this method. Upon applying the method the list of user defined slots is displayed, from which the slots to be removed are selected. At least one slot must be selected, otherwise an error message is displayed. Multiple slots may be removed in one method application.

4.2.4 Get properties from the databank

A databank is available with some basic component properties. The properties that it contains are: component name, absolute formula, CARN (Chemical Abstracts Reference Number), ASPEN name, molecular weight (in kg/kmol) and the atmospheric boiling point (in Kelvin). If the component name is recognised, then the data from the databank are added to the appropriate slot in the object with status *obtained from databank*. If the object slot has status *entered by user*, then the slot remains unaltered.

4.2.5 Estimate properties using group contribution method

Some physical properties may be estimated by using a group contribution method. This requires the structural formula of the component, which may be entered directly

to the slot `Struct_formula` in the object. However if this slot has status *unassigned* when this method is applied, then the user is presented with a “molecular typewriter” (see figure 4.2), using which the structural formula may be built up. Some complete molecules are also present on the typewriter.

The Molecular Typewriter

This JavaScript application lets you estimate basic properties of simple organic chemicals by constructing the molecule from its constituent groups.

Please enter the structural formula for component *Acetic_acid*, absolute formula C2H4O2

Group Contributions	Properties	Complete Molecules
CH₃ CH₂ CH C (CH₂)₂ COOH CHO OH CO O COO S OH-(benzene ring) >C= CH= CH₂= -C= O= CH(triple bond) C(triple bond) Start Cyclic Ring End Cyclic Ring F Cl Br I CN NH₂ NO₂ aromatic C₆H₅	Click Clear to start or restart. CH₃COOH Clear View Properties	CH₄ H₂O H₂S N₂ O₂ H₂ Cl₂ Br₂ I₂ F₂ HCl HBr HI HF C₆H₆ NH₃ HCN CO₂ CO

Figure 4.2: Entering the molecular formula using the molecular typewriter

The properties are estimated using Joback’s group contribution estimation methods. These methods work by giving values to the structural constituents of the component. These values are then summed, and put into formulae to estimate various properties. Different constituent values are used for estimating different properties. The program used to calculate these properties was developed by Haifa Wahyu [52] as part of her PhD.

The properties estimated using this method are:

- Molecular weight (kg/kmol)
- Atmospheric boiling point (K)
- Heat capacity polynomial coefficients (kJ/kmol/K)

- Critical temperature (K)
- Critical pressure (bar)
- Critical volume (cm³/mol)
- Free energy of formation (kJ/mol)
- Enthalpy of formation (kJ/mol)

Once the properties have been estimated, they are displayed and the user has the choice over which property values to add to the object, see figure 4.3.

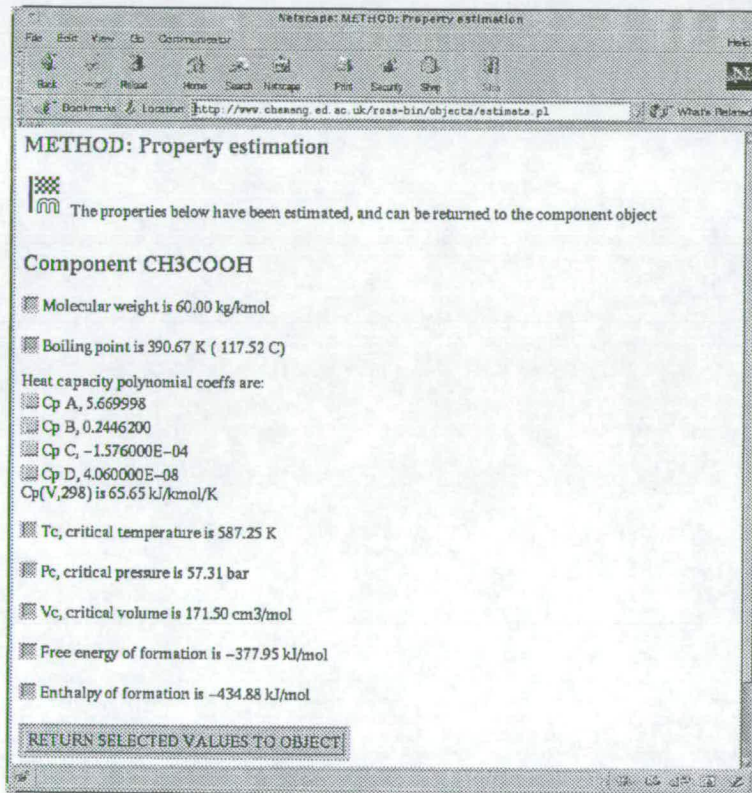


Figure 4.3: Returning component properties from estimation method

Any selected data are returned with status *estimated*. If any slot has a status value of *entered by user* or *obtained from databank*, then the slot is not overwritten.

4.2.6 Create a spreadsheet containing the object data

Applying this method creates a spreadsheet containing the object data. For each data slot the name, value, description and status are extracted and written to a SYLK

format spreadsheet. For any given object, this spreadsheet will only be created once. Any further application of this method to the object will result in the same spreadsheet being displayed, as the object data are unaltered. If the data were changed then a new object would be created. The spreadsheet is stored as a “child” of the object.

4.3 Mixture

The mixture object is designed to group together component objects for ease of reference. When created it only contains the names of the constituent component objects, with no further data slots.

The methods for a mixture object are:

- Update the object to register any manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Add component(s) to the object
- Remove component(s) from the object
- Create a spreadsheet containing component data
- View VLE data on selected components

The first three methods are identical to the first three methods in the component object, and have been described in sections 4.2.1, 4.2.2 and 4.2.3. Indeed, all objects have at least these three methods present. They will not be discussed further in this chapter, unless there are significant differences in applying the methods to different object classes. The remaining methods are described below.

4.3.1 Add component(s) to the object

More components may be added to the mixture object by applying this method. These components are selected from a list of all component objects in the user’s workspace

not already present in the mixture object. The only limit on the number of components that may be added is that at least one component must be selected from the list. There is no upper limit, other than the number of components available. The same component object may not be present twice in any mixture object.

4.3.2 Remove components(s) from the object

As well as adding component objects to the mixture object, components may also be removed. Applying this method presents the user with a selection list displaying the component objects present in the mixture. The user may then choose the components that are to be removed. At least one component must be selected from the list. At the upper limit, all components present in the mixture may be removed by applying this method.

4.3.3 Create a spreadsheet containing component data

This method is comparable to the method of the same name in the component object, described in section 4.2.6. Instead of extracting data from a single component object, this method extracts data from all component objects in the mixture and puts it in a SYLK format spreadsheet. This spreadsheet is stored as a “child” of the object.

4.3.4 View VLE data on selected components

This method enables the user to view binary Vapour Liquid Equilibrium (VLE) data for two components at a user specified pressure. Upon applying the method, the user is presented with a selection list containing all the mixture components. The two components of interest are selected from this list. The pressure for which the data will be generated must also be given. This method then creates an ASPEN input file with the relevant data as specified by the user, extracting the ASPEN databank component names from the slot `Aspen_name`. If this slot is empty then an error message is displayed. The ASPEN simulation engine is used to generate the data, using the UNIQUAC physical

property prediction method. The data is returned to the user in the form of a table in a SYLK format spreadsheet. An example input file used to calculate VLE data for benzene and toluene may be found in appendix E.1. An extract of the spreadsheet generated is shown in figure 4.4.

This data was generated at a pressure of 20 bar using the UNIQUAC property estimation method

Mole Fraction benzene	Vapour Mole Fraction benzene	Vapour Mole Fraction toluene	Liquid Mole Fraction benzene	Liquid Mole Fraction toluene	Temperature (deg C)
0.000	0.000	1.000	0.000	1.000	263.229
0.025	0.025	0.975	0.015	0.985	262.478
0.050	0.050	0.950	0.030	0.970	261.714
.	.	.	.	.	.
0.950	0.950	0.050	0.911	0.089	225.446
0.975	0.975	0.025	0.954	0.046	224.021
1.000	1.000	0.000	1.000	0.000	222.540

Figure 4.4: The output from calculating VLE data on Benzene and Toluene at 20 bar.

4.4 Stream

The stream object contains the data for a process stream. The data slots are grouped in two separate areas: the component data slots which contain the components present in the stream and their molar flows; and the other data slots which contain general stream property slots: the stream temperature and pressure. The stream object also contains a mixture object, which in turn contains all the component objects of the stream. This linked mixture object remains the same for any child objects of the stream, unless the component list is altered, upon which a new mixture object is created and linked. Components are displayed as being contained within a phase. If a stream object is created with no explicit phase information, then a single phase stream is created with phase name: `default`. The status of this phase slot is *estimated* and may be altered by the user. If the stream object contains more than one phase, then an identical list of component objects is used for each phase.

Slots are present to enter the molar flows (in kmol/hr) for each of the components in the

stream. If the component object has its molecular weight slot defined, then JavaScript functions may be used to view the entered molar flows for a variety of flow basis, e.g. kg/s, te/hr, etc.

Component Data Slots

Each slot has four attributes:
Component Name, Molar Flow, Alternative Flow Representation, Status

- Component Name: The name of the component
- Molar Flow: The molar flow of the component. Please type the component flow into this box
- Alternative Flow Representation: For JavaScript enabled browsers, this box will display the flow in alternative units, if molecular weight data is available for the component NB Do NOT type here directly
- Status: Unassigned for an empty slot entered by user for a filled one

Phase:	Component	Flow		Status
		kmol/hr	kmol/s	
MIXED	100_242_benzene_1_01.html	2.00	0.00127499	entered by user
	100_242_hydrogen_1_01.html	4.59	0.00127499	entered by user
	100_242_methane_1_01.html	4.59	0.00127499	entered by user
	100_242_toluene_1_01.html	0.20	0.00055555	entered by user
	100_242_diphenyl_1_01.html	4.80	0.00133333	entered by user

Other Data Slots

Temperature:	300 K	Status: entered by user
Pressure:	1000000 N/m ²	Status: entered by user

Figure 4.5: Data slots for a stream object

The methods for a stream object are:

- Update the object to register manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Add component(s) to the object
- Remove component(s) from the object
- Create a spreadsheet containing the object data
- Add a new phase to the stream

The first three methods have been discussed in sections 4.2.1, 4.2.2 and 4.2.3 respectively. The remaining methods are described below.

4.4.1 Add component(s) to the object

Applying this method to a stream object is very similar to applying the method of the same name to a mixture object as discussed in section 4.3.1. Upon the method application, the list of components in the user's workspace not present in the object is displayed, and the user selects the component objects to add. The selected objects are then added to each phase in the stream object.

4.4.2 Remove component(s) from the object

As well as adding components to the stream object, components may also be removed. This method is almost identical to the method of the same name discussed in section 4.3.2, which is applied to mixture objects. The only difference is that here the selected components are removed from all phases in the stream object.

4.4.3 Create a spreadsheet containing the object data

This method is similar to the equivalent method in the mixture object, discussed in section 4.3.3. The spreadsheet generated consists of two parts. The first part is the component property section, which is identical to the spreadsheet generated from the mixture object and consists of the component properties for the components in the stream object. The second part of the spreadsheet contains the molar flows and mole fractions for the component flows for each phase in the stream.

4.4.4 Add a new phase to the stream

Applying this method adds a new phase to the stream object. The new phase contains the same components as the other phases present in the stream. To distinguish the new phase from the other phases, it must be given a name. The name of the phase added is automatically selected to be default and has status *estimated* so that it may be changed if desired. If this phase name is already used by a phase in the stream,

then a phase name of `default_1`, `default_2`, etc. is chosen until an unused name is obtained.

4.5 Reaction

The reaction object is designed to store details of a reaction. Like the stream object the data slots are separated into two areas: the component data slots and the other data slots. The component data slots contain the components present in the reaction, and their reaction coefficients. Again like the stream object, when the reaction object is first created these components are used to create a mixture object, linked from the reaction object.

The other data slots are for: temperature, Gibbs free energy at 298K, enthalpy of reaction at 298K and the Gibbs free energy at the specified temperature.

The methods for a reaction object are:

- Update the object to register any manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Add component(s) to the reaction
- Remove component(s) from the reaction
- Calculate reaction properties
- Clear other data slots

The first three methods in the above list have been described previously in sections 4.2.1, 4.2.2, 4.2.3. Also the method “remove component(s) from the reaction”, is so similar to the equivalent method for a mixture or stream object (see sections 4.3.2 and 4.4.2) that it will not be described here. The remaining methods are discussed below.

4.5.1 Add component(s) to the reaction

The user may add more components to the reaction by applying this method. The difference between applying the method to a reaction object, and applying the equivalent method to stream and mixture objects (see sections 4.4.1 and 4.3.1) is that here only components that have all their constituent slots filled, i.e. no slots have status *unassigned*, are displayed in the list of component objects which may be added to the reaction. The reason for this is that when applying the method “Calculate reaction properties”, to a reaction object, component data are extracted from each of the reaction components and used in this method. This is the only point in the system where it is required for an object to have its constituent slots filled. The component data needed may be obtained by applying the relevant method to the component object.

Other than this exception the procedure is the same as for adding component objects to mixture and stream objects. At least one component must be selected from the list, which is then added to the reaction object.

4.5.2 Calculate reaction properties

Applying this method calculates several reaction properties using a program developed by Wahyu [52]. The properties calculated are:

- Free energy of reaction at 298K (kJ/mol)
- Enthalpy of reaction at 298K (kJ/mol)
- Free energy of reaction at a specified temperature (kJ/mol)

The specified temperature at which the free energy of reaction is calculated, is that temperature given in the reaction object. If no temperature is given in the `temperature` slot, then only the first two properties are estimated. In order to calculate these properties, data are extracted from the component objects taking part in the reaction. If the component objects are not complete, then an error message is displayed. An error

message will also be displayed if the reaction specified is not feasible, i.e. elements being created or removed during the reaction.

The user has the option of returning the data to the reaction object if satisfactory. Any data returned to slots in the reaction object have status *estimated*.

4.5.3 Clear other data slots

If the user has applied the method “Calculate reaction properties”, and returned the generated data to the object, then the “other data slots” of the reaction object will be filled. If the user wishes to estimate the reaction properties at a different temperature then this method may be applied which clears the temperature slot, and the other three slots containing reaction property information. It does not affect any user defined slots, or the component slots. This method saves the user from having to locate, or indeed even create, a reaction object with no temperature defined and an identical reaction.

4.6 Unit

The unit object is intended to hold data for an item of equipment performing a unit operation. When created, the object contains slots for a description, the unit feed and product streams, and the name of the unit. The description slot is used to enter a natural language description of the unit which is then parsed and used to both create slots and fill the created slots. The feed and product stream slots are used to store the feed and product streams of the unit. The names of the streams are displayed, and hyperlinked to the stream object. Note that these names need not be the same as the object names, and are normally different, e.g. feed, 1, product, etc. They are intended to be used from within the object for ease of reference. No status is displayed for these streams. The unit name slot is a standard object slot.

The methods for a unit object are:

- Update the object to register any manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Create an ASPEN model of the unit
- Use the latest version of referenced objects

4.6.1 Update the object to register any manual changes

As described in section 4.2.1, this method checks to see if any changes have been made to the object, and updates the object accordingly, changing the status values of the slots where appropriate. In the unit object it also performs a further task. If data have been entered into the description slot, then this description is parsed and data extracted. This is discussed in detail in appendix A.

Briefly, the description is examined for a key word that indicates the function of the unit. If the unit function is determined, then the description is again parsed. This time it is examined for a list of key words peculiar to the unit function. These data are used to create and fill slots in the unit object.

4.6.2 Add a slot to the object

As well as adding a data slot to the object as described in section 4.2.2, this method may also be used to add either a feed or product stream to the unit. This is accomplished by entering either feed or product in the Slot Name box (see figure 4.1). In the Value box the name of the stream object should be entered. Finally in the Description box the name by which the stream object will be referred to in the unit object needs to be entered. This will then add the stream to the unit object. If the stream object given does not exist in the user's directory then an error message is displayed.

4.6.3 Remove a user defined slot from the object

This has been discussed previously in section 4.2.3.

4.6.4 Create an ASPEN model of the unit

Applying this method creates an ASPEN model of the unit. The creation of ASPEN models is described in detail in appendix H. Briefly there are a number of stages. Firstly the component names are extracted from the feed and product streams, and their ASPEN names obtained. Next the unit connectivity is described by listing the feed and product streams of the unit, together with the unit name. After this the feed stream details are given and finally the unit parameters are extracted and added to the model. The parameters required differ depending upon the unit function. The list of required parameters for each function is given in appendix H.4.

4.6.5 Use the latest version of referenced objects

Both the feed and product streams are objects in their own right. This may be true of other slots in the unit as well, e.g. reaction slots. These objects may be updated, e.g. by adding new slots. In order to propagate these changes through to the unit object this method may be applied. It changes any selected hyperlink, linking to an object, to link to the latest version of the object. By latest version it is meant the last object created, either directly or via an intermediate object, from the object currently referenced by the hyperlink. For example consider the history tree in figure 4.6. The latest object created in this family is `100_242_mix1.05.html`. Thus if a unit object containing the object `100_242_mix1.01.html` in a slot were to have this method applied to it, then the old object would be replaced by `100_242_mix1.05.html`.

Upon applying the method, the user is presented with a selection list containing all of the objects that may be updated. The user may either select individual objects from this list, or update all the links in the object.

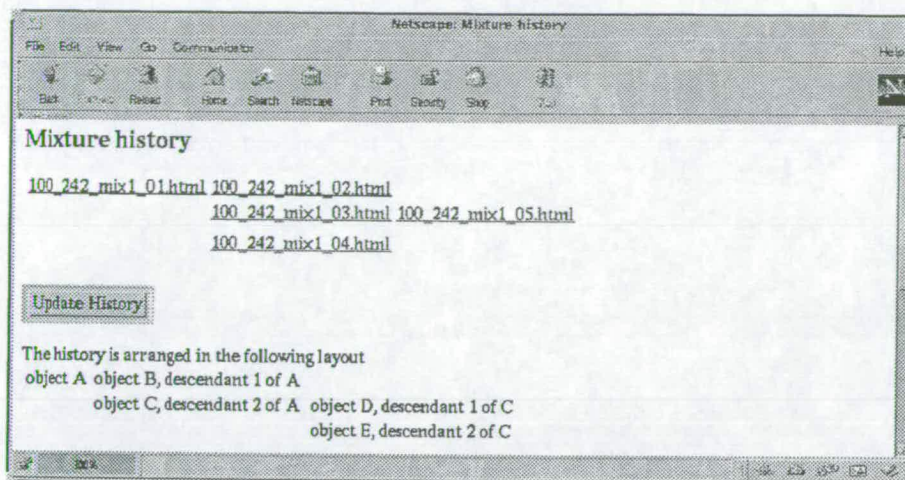


Figure 4.6: A Sample History Tree

4.7 Topology

The topology object is designed to hold all of the necessary information for a process. It contains slots for the topology name and purpose, both of which are intended to be used by the user for documentation only. In a similar manner to the unit object, the topology object also contains slots for the topology, feed and product streams, as well as any units. The topology itself lists the streams and units, and shows their connectivity. The process is specified by means of a natural language description which is parsed and used to create stream and unit objects. Any topology feed streams, product streams, and units are entered into the appropriate slots in the object. When the topology object is created, a mixture object must be specified. The components in the mixture object are used for the components in the stream objects created by the parsing. The lack of a graphical input is compensated somewhat by the fact that a three dimensional VRML diagram is generated from the parsed description, using a program written by Hallet [25].

The methods for a topology object are:

- Update the object to register any manual changes
- Add a slot to the object
- Remove a user defined slot from the object

- Decompose single unit into two or more units
- Separate units into different topology objects
- Combine topology objects together
- Change stream/unit target
- Use latest versions of objects
- Create a spreadsheet model
- Return data from a solved spreadsheet model
- Create an ASPEN model
- Create a HYSYS model
- Get data from an ASPEN model

4.7.1 Update the object to register any manual changes

This method serves two purposes:

1. It checks to see if any changes have been made to the object data slots, and updates the object accordingly, changing the status of the relevant slot, if required (see section 4.2.1).
2. It parses a natural language description of the process, if given, and extracts the relevant data to create stream and unit objects, and the topology.

The parsing works by identifying different categories in the description. The categories recognised are:

- Unit identifiers comprised of a single capital letter, followed by between one and three numbers, e.g. *B3*, *R105*, etc.
- Stream names which consist of whole integers, e.g. 5, 3, etc.
- Operation names, given as a list of key words.

Table 4.1: Operation names recognised by parsing

Operation Name	Unit Function	No. Inputs	No. Outputs
feed	feed stream	-	1
mix	mixer	2	1
reac	reactor	1	1
sepa	separator	1	2
spli	splitter	1	2
cont	contactor	2	2
heat	heater	1	1
cool	cooler	1	1
vapo	vaporiser	1	1
cond	condenser	1	1
strip	stripper	2	2
absorb	absorber	2	2

The natural language parser is based on an algorithm developed by Professor J. W. Ponton, which recognised stream and operation names. The original algorithm was extended such that unit identifiers were also recognised, and that HTML objects were created for the parsed streams and units.

The operation names recognised, together with their unit function and the expected numbers of input and output streams, are listed in table 4.1. If stream identifiers or unit names have been omitted in the description then the parser will generate them.

Only the stem of the word checked must match, e.g. `reac` will match against `reacted`, `reacts`, or `reactor`. The matching is case insensitive.

The description is parsed, and any unit identifier, stream name or operation name is extracted. If the word “twice”, is mentioned, then it is assumed that the most recently matched operation is to be repeated. If the keyword `feed` is matched, then the next word in the description is examined to see if it is a stream identifier. If not one is generated and inserted into the extracted words. Stream identifiers are generated by first of all extracting all of the stream identifiers from the description. If any new stream identifiers are required, then the identifier 1 is checked to see if it has been used, and this is incremented until an unused integer has been obtained. The reason for the addition of the stream identifier when “feed” has been matched may best be explained

by the following example. Consider the following two extracts from a description:

1. feed and 3 mix
2. feed 3 and recycle are mixed

In the first example, the feed stream has not been given an identifier, whereas in the second example the identifier 3 has been used. However without adding in a feed identifier when required, both examples will parse to: "feed 3 mix", which is not what is meant in the first example. Adding in a feed stream identifier solves this problem.

The parsing is best illustrated by parsing the following description:

Feed stream and 2 are mixed in M102, reacted twice and separated to give 2 and the product stream.

The initial parsing extracts the following:

feed 1 2 mix M102 reac reac sepa 2

There is a particular order that stream names and unit identifiers must be given in the description. Any input stream identifiers must be given first, followed by the unit function, followed by either the unit identifier or the output streams. It does not matter in what order the unit identifier or the output streams are given in.

At this point if any stream or unit identifier is required but has not been given in the description, then it is added in to the list of extracted words. This is to cover the eventuality of the output stream of one unit being the input stream of another unit. Unit identifiers are generated by using *B1*, *B2*, etc. checking to ensure that the generated identifier has not been given in the description.

Applying this to the example gives:

```
feed 1 2 mix M102 3 reac B1 4 reac B2 5 sepa B3 2 6
```

The next stage is to organise the data in a structured manner. Each unit is written, with its identifier, to a line. The following line contains the input streams, two or more spaces, and then the output streams.

Organising the example gives the object topology:

```
feed
1
mix      M102
1 2      3
reac     B1
3        4
reac     B2
4        5
sepa     B3
5        2 6
end
product
6
```

There is an `end` statement present after the unit operations, but before the product streams. The topology is required in this format to generate a VRML diagram of the topology.

At this stage in the parsing the user is presented with the parsed description in the above format. This is also the format of the approximate topology selected from the model generation manager (see section 3.3.1). If the description has not been parsed as the user wished, it may be altered. The final task is to create stream and unit objects for each of the units and streams in the topology.

Stream and unit objects are only created if no suitable objects already exist. Suitable objects are defined as being in the same family, and representing the same named stream or unit. For example, if the stream identifier 4 were used in the topology of session identifier 99_14, then any streams in the stream family 99_14_stream4 would

be suitable to be used. The last created stream in this family would be used. A similar process is carried out for units.

If stream or unit objects have to be created the following procedure is implemented. Stream objects are created with a single phase, containing the same components as the topology mixture object. Unit objects are created with data slots appropriate to their purpose. The list of data slots for each function is identical to those units created after parsing the unit description for the various functions (see appendix A).

Apart from the Name and Function slots which are filled, all other slots are created empty with status *unassigned*.

4.7.2 Add a slot to the object

This method is the same as discussed in section 4.2.2. Note that it is not possible to add feed or product streams to the topology in the same manner as adding streams to a unit object as described in section 4.6.2.

4.7.3 Remove a user defined slot from the object

This method has been described in section 4.2.3.

4.7.4 Decompose single unit into two or more units

At the initial stage of a design, the process itself is only vaguely defined. It may consist of feeds and products, but the stages in between have not yet been considered, i.e. a black box model. Over time, the process will be firmed out. At each iteration, several of the unknowns in the design will be worked out, and the black box can be replaced, either with unit operations or smaller black boxes. Replacing a large unknown block with several smaller blocks is known as decomposition, see figure 4.7.

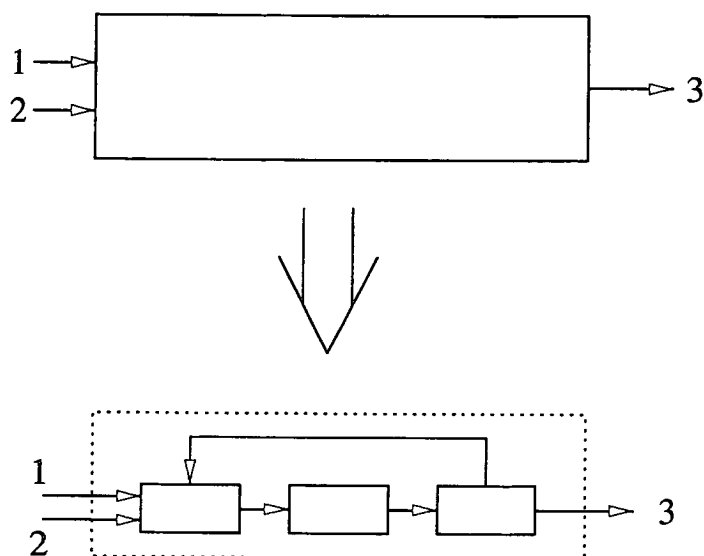


Figure 4.7: The decomposition of a black box model

Applying this method to a single unit topology, i.e. a topology containing only a single unit, results in the original topology being removed in the new object. A new topology description may then be entered. The original feed and product streams remain in the new topology object, and must be contained in any new topology description entered.

4.7.5 Separate units into different topology objects

Prior to decomposition, the units in the topology must be separated so that each unit object is contained in a topology object of its own. Applying this method creates several topology objects, each containing one of the units in the original topology. Only topology objects with multiple units may have this method applied to them.

4.7.6 Combine topology objects together

One possible consequence of decomposition is that the process model may comprise of several topology objects. Applying this method allows the user to join these objects together to form one single topology object. Upon applying this method the user is presented with a selection list displaying all the topology objects present in the user's workspace. Only one topology object may be combined with the present topology

object at each method application. An important point to note is that two topology objects may only be combined if the same stream object is used as a product stream of one topology object and as a feed stream to the other. If this is not the case then an error message will be displayed upon the selection of an invalid object.

4.7.7 Change stream/unit target

Using this method, it is possible to change the target of a unit or stream name in the topology. The target of a name is the object to which the name is hyperlinked, e.g. if the unit named B1 were hyperlinked to the object 99_342_unit1_01.html, then the target of B1 would be 99_342_unit1_01.html. After changing the target, the name of the stream or unit will remain the same, but the object to which it is hyperlinked will be changed.

Upon applying this method the user is presented with a selection list displaying all of the names of the streams and units in the topology, together with the topology mixture. From this list one item must be selected, otherwise an error message is displayed. After selecting the desired stream, unit or the mixture, the user is then shown a list comprising of all created objects of the same class as the target of the item selected, present in the user's workspace. One of these objects must be selected as the new target for the name chosen from the previous list. It is then linked from the new topology object, replacing the old object.

Note that if the topology mixture is selected to be changed, then each stream in the topology will have its components changed to the new mixture selected, and each unit changed to use the new streams.

4.7.8 Use latest versions of objects

This method has been described in section 4.6.5.

4.7.9 Create a spreadsheet model

The user may create a spreadsheet model, in SYLK format, of the topology by applying this method. The model takes as inputs component flows for the feed streams, and parameters for the various unit operations. It displays as its output a stream table giving component flows for each of the streams in the model. Applying this method is identical to creating a spreadsheet model from the model generation manager.

The spreadsheet created has three distinct sections: a component area, a data input area, and a stream table. The component area contains the component data for each of the components in the model. This is extracted from the component objects contained in the topology mixture object. The data input area is where the feed stream component molar flows, and the unit parameters are entered. Where possible, values are extracted from the relevant objects and entered here. The remainder of the spreadsheet is write protected. The final area, the stream table, displays the component molar flows for each of the streams in the model. A sample spreadsheet is given in appendix I.2.4.

The spreadsheet is generated from the model topology. This is used to form a set of linear equations which are solved in the spreadsheet. The equations generated for each unit type are displayed in appendix B. Many of the unit operations require parameters, which are listed below:

- **Feed Stream** - Although not a unit operation, feed streams require the molar component flowrates of each of the components present. The information required is extracted from the relevant feed stream to the topology.
- **Reactor** - The data needed for a reactor are the stoichiometric coefficients for the reaction, the key component, and the conversion of the key component. The reaction data are extracted from the reaction object in the unit. If only one reaction is specified, then it should be in the slot `Reaction`. If more than one reaction is present in the unit object, then the reactions should be in the slots `Reaction_1`, `Reaction_2`, etc. The same convention is followed for naming slots for other data when the unit contains multiple reactions. The conversion (in

slot Conversion) and key component (Key_component) are extracted from the unit object modelling the reactor. If a reaction object is not present, then the coefficients are left blank. If no key component is specified, then the user is prompted to select one from the topology mixture components. The conversion is constrained to be between 0 and 1. If a unit object contains more than one reaction, then the correct number of reactions are automatically inserted in the spreadsheet.

- **Separator** - This unit operation requires the mole fractions of each of the components in the feed stream going to specified output streams. Fractions are required for all but one output stream. If the feed and product stream objects have component flows defined, then the component split fractions are calculated from these. Otherwise they are left blank for the user to complete. The split fractions are constrained to be between 0 and 1.
- **Splitter** - The only parameter required here is the fraction of the input stream going to the specified output stream. This is constrained to be between 0 and 1. If the feed and product stream objects for the unit object have component flows defined, then it may be calculated. Otherwise it is left blank.
- **Contactor, Stripper, Absorber** - Each of these unit operations require the molar component split fractions for the combined input for that component going to a specified output stream. Fractions are required for all but one output stream. As with the separator, if the input and output stream objects have component flows entered, then the component split fractions may be evaluated. Otherwise the user must input them.
- **Heater, Cooler, Condenser, Vaporiser** - The spreadsheet only performs a mass balance. No heat balance is carried out. Thus in the spreadsheet the output streams of these units are identical to the input streams.

The linear equations set up from these units are solved using Gaussian elimination. The Gaussian elimination code is generated automatically, and held in hidden cells in the spreadsheet.

Gaussian elimination works by taking the first equation of the equation set, and di-

viding the equation by the coefficient of the first variable so that the coefficient of this variable, the elimination variable, is one. Multiples of this equation are then added to or subtracted from the second and subsequent equations in the set to remove from them this variable. This is repeated with a variable in the second equation to remove this variable from the third and subsequent equations, and so on until the final equation has only a single variable with a coefficient of one. This is then back substituted to find solutions to the remaining variables. The problem with this method is if the coefficient of the elimination variable is zero, then the equation will be divided by zero. This is known as a “zero pivot”. More sophisticated systems reorder the equation set such that this does not happen. However with the spreadsheet model, the order of the equations is predetermined. With the spreadsheet, zero pivots arise when component recoveries, and split fractions are either zero or one. To remove this risk, both of these variables are constrained to be greater than zero, and less than one.

Also constrained to be between zero and one are the fractional conversions of the specified reactions. These implicit constraints only show up in the spreadsheet, which potentially creates inconsistencies between the results calculated using the spreadsheet, and those calculated using other tools. Thus the user may have two differing sets of results generated from the same data. It is up to the user to decide if the results generated using the spreadsheet are accurate enough for the purpose intended. To avoid any inconsistencies, the user would need to obey the implicit constraints of the spreadsheet when entering data in an object.

4.7.10 Return data from a solved spreadsheet model

Once a spreadsheet model has been generated, it may be used to calculate the molar component stream flows for the topology. Applying this method enables the user to return data from a solved model to the relevant stream objects. The solved spreadsheet must first be saved in CSV (Comma Separated Values) format to enable data to be extracted. Upon applying the method, the user is prompted for the location of the spreadsheet. Either a local file name or a URL may be given. Note that the topology from which this spreadsheet was generated must be identical to the topology of the

object to which the data are being returned. Otherwise an error message is generated.

Only the molar component flows of the streams are extracted from the spreadsheet. New streams are created to hold this data, children of the stream objects of the topology to which the method is being applied.

No unit parameter data are extracted from the spreadsheet. However, as the stream objects have changed, new unit objects are required to store the changes in the unit feed and product streams. These unit objects are children of the unit objects of the topology to which the data are being returned.

4.7.11 Create an ASPEN model

AspenPlus (usually shortened to ASPEN) is a simulation package which enables models of processes to be created and simulated. All simulation models need a number of items of information: the component names, the connectivity of the units, process feed stream data, and the individual unit types and parameters.

Applying this method extracts information from various objects and writes it in an ASPEN model. There are a number of formats in which the model may be written. The most convenient to generate is the input file, which is a simple text file. One major drawback though is that this format cannot be loaded into the ASPEN GUI, Model Manager, unlike the other ASPEN formats, backup or quick restart. However it is possible to use the ASPEN simulation engine to generate a backup format file from an input file. Upon the successful application of this method, an input file is generated. For local users, this is transformed into a backup file. This may not be done for all users due to ASPEN licensing restrictions.

This method requires a number of different parameters, e.g. specific slots must be defined. If any of these parameters are missing then the input file cannot be generated. Rather than stop the method execution when a single error is encountered, this method continues on as much as possible. All generated error messages are then collated and

presented to the user.

The parameters required to create the ASPEN model are given in appendix H.

4.7.12 Create a HYSYS model

HYSYS is another simulation package, and the writing of a HYSYS model has similar requirements to that of ASPEN, i.e. the component names, unit types, connectivity and unit parameters must be obtained. Applying this method extracts the relevant data and uses the data to create a HYSYS file. The model is created in the form of HYSYS script files. Two script files are created: one containing the component information, and the other containing the feed stream, unit parameters and connectivity. This is necessary as there is no databank of HYSYS component names. Thus estimates must be made of their names by using the value given in the component name slot. If this name is not a HYSYS component, then HYSYS will display an error message as the script is being run, and stop executing the script. As two script files are used, it means that if a component name has been incorrectly estimated, then only the component information is affected. The user may enter the remaining component names manually and then run the other script file to finish loading the model.

This method requires a number of parameters, in much the same manner as the method used to create the ASPEN model (see section 4.7.11). Like that method, if any of these parameters are missing or are incorrect, then the model may not be generated. Instead of stopping the method execution when an error occurs, this method continues on as much as possible. Once finished all error messages are collated and presented to the user.

The parameters required to create the HYSYS model are given in appendix H.

4.7.13 Get data from an aspen model

Once an ASPEN model has been created, either by applying the method “Create an ASPEN model”, or by other means, it is possible to extract data from this model and either create a topology object and the associated stream and unit objects, or return data to an existing topology. The ASPEN model may or may not be simulated, i.e. the model does not have to contain results. Simulating an ASPEN model generates a number of files, only some of which are parsed here. There are also two methods of running ASPEN: either via a text input file run on the command line or using a graphical user interface. The files parsed here are:

- Backup file (.bkp extension) - This file is generated using the graphical user interface. There are two types of backup files: those that contain only the unit types, their parameters and connectivity, and those that contain this information together with the simulation results.
- Input file (.inp extension) - This is the text input file which contains the model information needed to run a simulation from the command line.
- Summary file (.sum extension) - The summary file contains the results of a simulation run from the command line.

Upon applying the method, the user is asked to type in the URL of one of these files. This file is then uploaded and the information extracted from it. The preferred option is the backup file, with simulation results. The same information is contained in a combination of the input file and the summary file for the same simulation. If the URL of either an input file or a summary file is given, then a check is made to see if the other file (with the same stem but different extension) may be uploaded from the same directory. This is not essential, and sufficient information may be extracted from either file to create a topology object and relevant unit, stream and component objects. If only an input file, or a backup file with no results, is given, then objects for the feed streams, and for the units may be created with the component flows and unit operating parameters. However no intermediate stream component flowrates will be available. Alternatively if only a summary file is given, then these intermediate

flowrates are available, but no unit parameters are obtainable.

Note that if this method is applied to a topology object that has a topology defined, then the topology of the uploaded ASPEN model must be identical to the topology already given, i.e. the unit functions, the unit names and the stream names must be the same. No checks are made on the components present, or on any of the object slot data in the original topology object to see if they conform to the given ASPEN model. If no topology has been given in the object then none of these checks are made.

When parsing data from either the summary file, or a backup file with results, then a large amount of data are obtained about the operating conditions of the units. In most cases this is included in slots in the unit object without parsing for specific slot data. However in certain cases this data are parsed to fill slots created when the object function was known. For a list of these slots see appendix A.

In extracting the data from the ASPEN model, the first task is to obtain the names of the components. These are checked against the list of component objects in the user's workspace. If an object exists for any component then this is used in the topology, otherwise a new component object is created. Next the stream data are obtained. This may be specified in the ASPEN model either in component molar flows or as component fractions of a total molar flow. It is not possible to parse stream information if it is given in mass flow. Also extracted at this point are the stream temperature and pressure. The final task is to extract information from the unit models. Not all ASPEN unit models are parsed here, and of the ones that are parsed, not all of the data are extracted. To write a system which could deal with all eventualities would be very complex! The ASPEN unit models recognised here are:

- **mixer** - This translates to a mixer unit, i.e. a unit object with the Function slot defined as mixer.
- **fsplit** - This is used to create a splitter unit object. It is assumed that only stream fractions are given for the output streams, which are written into slots in the unit. If stream flows are defined for the output streams then no data on the

split are written to the unit.

- **flash2** - This is modelled in a separator unit, which has its `Split_sharpness` slot set to semi-sharp.
- **rstoic** - This unit is modelled using a reactor unit. Each of the reactions listed here are stored in reaction objects. Either the key component and conversion or the extent of reaction is extracted from the data.
- **sep** - A “sharp” separator unit, i.e. the `Split_sharpness` slot is set to sharp, is used to model this unit. No object slots, other than the name and function, are set here.
- **dstwu** - This unit is also modelled with a “sharp” separator. It is used in ASPEN for shortcut distillation calculations. Data are extracted for the key components, their recoveries, the reflux ratio, and the operating pressure of the column.
- **radfrac** - This model is used in ASPEN to perform rigorous separation calculations. If the model has one input and two outputs, then it is assumed to be a “sharp” separator. However if it has two inputs and two outputs then it is modelled as an absorber. Note that for the purposes of checking to see if the object topology is equivalent to the ASPEN model topology, then the absorber is equivalent to either a contactor or a stripper.
- **heater** - This ASPEN unit is modelled using one of a number of different unit types. If a vapour fraction is given and set to zero, then a condenser is used. If the vapour fraction is given and set to one, then a vaporiser is used. Otherwise it is modelled as a heater. For the purposes of checking the equivalence of the object topology and the ASPEN model topology, heaters and coolers are considered equivalent.

4.8 Equation

The equation object is used to hold and manipulate a set of linear equations, generated from a topology object. The equations in this object are the same equations as those used to create the spreadsheet model (see section 4.7.9), only now they are visible to the user, rather than hidden. Equations are generated for the component molar flows in

the output streams of each of the units. The equations are displayed grouped together in the separate units. For this analysis, feed streams are considered to be units with one output stream. Equation types are discussed for the individual units in appendix B.

The equation set of n equations may be represented mathematically by:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

The information stored in the equation object consists of the $n \times n$ matrix, and vector of constants on the right hand side of the equals sign. Also stored are the definitions of the unknowns, x_1 , x_2 , etc.

There are two versions of an equation object, each containing identical information, although displayed differently.

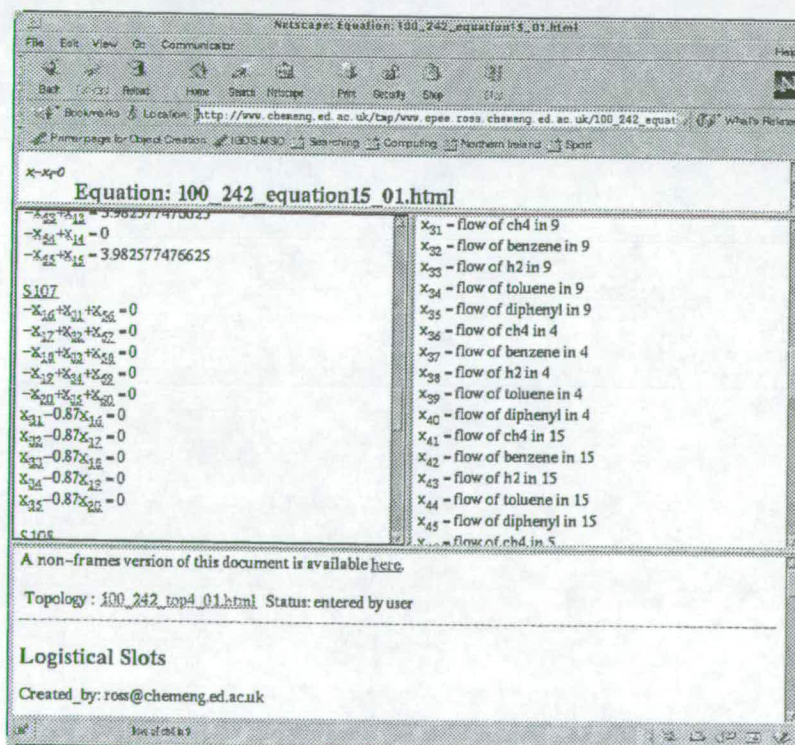


Figure 4.8: The frames version of an equation object

The default version of the object uses frames to display the information, see figure 4.8.

The equations are given in the frame to the left, with the variable definitions listed on the frame to the right. In the equation frame, all variable names are hyperlinked. Moving the cursor over any link results in the variable definition being displayed in the basis bar at the bottom of the browser. Activating this link displays the variable definition at the top of the right hand frame.

Also given as hyperlinks are the unit and stream objects from which the equations have been derived.

For those browsers that do not support frames there is also a version of the object which does not use frames. This contains the identical information, only this time the variable definitions are placed after the equations.

The methods for an equation object are:

- Update the object to register manual changes
- Add a slot to the object
- Remove a user defined slot from the object
- Solve the equations
- Edit the equations
- Delete some equations
- Add new equations/variables

The first three methods have been discussed previously in sections 4.2.1, 4.2.2 and 4.2.3 respectively. The remaining methods are described below.

4.8.1 Solve the equations

Applying this method solves the equations in the equation object, presenting the results in a stream table showing the component flows in each individual stream. If the equation set is under (too few equations) or over (too many equations) specified, then an error message is displayed.

4.8.2 Edit the equations

By applying this method, the user is able to edit certain equation variables of the equation set. There are two types of equation variable: the component molar flows (represented in the equation set by x_i), and other variables. These other variables represent quantities such as reaction co-efficient or component fraction, and will normally have been given a value in the equation. It is these other variables that may be altered by applying this method.

4.8.3 Delete some equations

The user may delete some of the equations in the object by applying this method. The equations are displayed, and then the user selects which ones to delete. At least one equation must be selected otherwise an error message is generated.

4.8.4 Add new equations/variables

If the user wishes to add new equations to the set, then this method may be applied. Upon applying the method the user is presented with a space where the new equations or variables may be entered. Existing variables may be used in the equations by typing x followed by the subscript number, e.g. x_2 for x_2 . New variables may be either defined separately, or defined in the equation where they are used. They are defined by simply enclosing the definition in a pair of braces, e.g.

$x_5 - \{\text{flow of methane in stream 5}\} = 34.5$

The user must also select to which unit, if any, the new equations refer to. A unit must be selected from a selection list containing all of the units, including feed streams, present in the equation object. A choice of "no particular unit" is also available.

Upon processing, this will create a new equation, with the text inside the braces being replaced with a symbol. This text will be recorded as the symbol definition. Multiple equations and definitions may be given at once, although each new equation or stand alone definition must be on a new line.

Any equation given is rearranged such that the unknowns are on the left hand side of the equation, and any constant on the right hand side. All of the elements in the equation may be edited by applying the method, "edit the equations".

Chapter 5

Application Programmer's Interface

5.1 Introduction

The previous two chapters have described the creation of objects on a WWW server. It was also shown how these objects could be manipulated on the server via a WWW browser. This chapter will describe how objects may be created and manipulated on the WWW server using Fortran programs. This is achieved by means of an Application Programmer's Interface (API), which consists of a series of Fortran structures to mirror the standard objects, and a set of Fortran subroutines. The combination of these structures and subroutines allows objects to be created on the server, and methods applied to them. There are no differences in the objects created via the API or via the browser.

The API provides an alternative way of interacting with the system to the graphical interface of the browser. It enables a user to sequentially apply methods and extract data from objects without having to repeatedly press buttons on a browser. The API routines may also be called in the program of a user, e.g. to apply a method to obtain physical property data for a component.

Fortran was selected as the language in which to write the API to demonstrate how

a language with no built-in WWW related features may be used to interact with the system developed.

The composition of the structures and the format of the subroutine calls are described in the following sections. The chapter concludes with an example program using API structures and subroutines to illustrate their use.

5.2 Implementation

Using a WWW browser, objects are created and manipulated in the following manner: a form is submitted, and a CGI script run which processes the form data, executes the method code, and returns the new object to the browser.

With the API, objects are created and manipulated in a similar manner with a Fortran program. If a call is made to an API subroutine, which requires the creation or manipulation of objects, then data are passed to a Perl script. This script runs the same CGI script as would be run by the equivalent form submission. The method code is executed, and the data returned to the Perl script, which returns the data to the Fortran program. This is illustrated in figure 5.1.

Implementing the API in this manner limits the code that needs to be distributed in order to use the API outside of the local filesystem. It also means that the objects created using the API are identical to those objects created using the WWW interface.

5.3 Fortran Structures

To complement the standard objects, several Fortran structures have been written for use with the API. These are described in detail in appendix G.1. To aid understanding of the API subroutines, a typical structure will be described below.

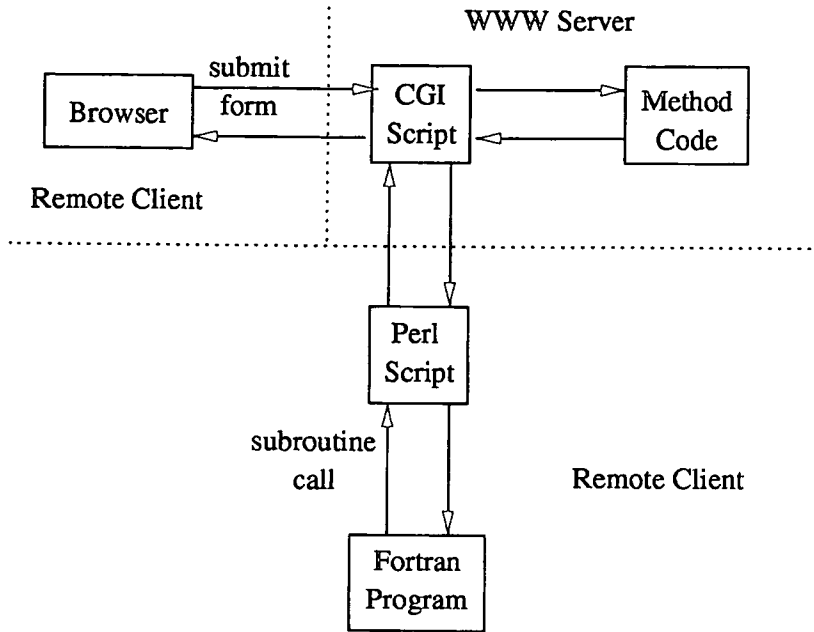


Figure 5.1: Method execution on the WWW server from remote clients

type component

```

!first of all for the arrays
type (slot) :: data_single(www_max_slots), log_single(www_max_slots)
type (multiple) :: data_multiple(www_max_slot_mult), &
    log_multiple(www_max_log_mult)

!need the url and address of the structure
character*200 :: url, address

!now for the www_hash lists
integer :: data_index(www_hash), log_index(www_hash)

!need to check if the object has been initialised
logical :: initialised

```

end type component

The object data are held in several arrays, divided among single and multiple types, and logistical and data slots. Each of these arrays are composed of a variation of the type slot. The type multiple consists of an array of the type slot. A slot itself consists of a character string for the name, and character arrays for the slot attributes and values.

Other elements of the structure include fields for the URL and address of the equivalent HTML object, if one exists. The `data_index` and `log_index` arrays are used to enable the access of slot arrays by slot name rather than index. Before a structure is used it must be initialised by setting all data fields to either zero or null strings. The initialised field records if this has been done.

5.4 Fortran Subroutines

The other constituent of the API, along with the structures, are the subroutines used to manipulate structures and objects. These are discussed briefly below. They are described in detail in appendix G.2.

- `www_setup_api` - This subroutine starts a session for a user, and performs many of the same tasks as when a user starts a session in a browser.
- `www_apply_method` - By calling this subroutine, the user is able to apply methods to objects that exist in the user's server workspace. This requires a code to indicate the desired method to apply. The codes are listed for all methods in appendix G.3.
- `www_write_html` - An HTML version of a Fortran structure is written to the server workspace by this subroutine.
- `www_get_obj_from_html` - This is the opposite of the `www_write_html` subroutine, in that it creates a Fortran structure from an HTML object.
- `www_initialise` - Before a Fortran structure can be used, it must be initialised.

This subroutine initialises the structure by adding the standard slots for the equivalent object class to the structure.

- `www_set_slot` - This subroutine may be used to add data to a slot in one of the Fortran structures. The advantage of using this subroutine, rather than manipulating the arrays directly is that the slot name is used in this subroutine, rather than the slot index.
- `www_write_xml` - A file containing an XML version of a Fortran structure is written by this subroutine.
- `www_add_slot` - By adding slots to the structure using this subroutine, the `www_set_slot` subroutine may be used to access the slots.
- `www_remove_slot` - As with the previous subroutine, by using this subroutine to remove slots from the structure, the `www_set_slot` subroutine may be used to access the remaining slots.
- `www_add_comp` - This subroutine adds a component to a structure, so is only relevant to mixture, stream and reaction structures.
- `www_get_mult_slot_attribute` - An attribute of a multiple slot may be accessed using this subroutine.

5.4.1 Overloaded Operators

Three overloaded operators have been defined to enable easier access of slot, component and attribute data. These operators are:

- `.SLOT.` - This is used to access a slot from an object, and returns a type slot. It is used in the following manner:

```
object .SLOT. slot_name
```
- `.COMP.` - This operator returns a slot type representing a component from either a mixture, stream or reaction object. It is used in the following manner:

```
stream .COMP. '00_23_ammonia_1_01.html'  
stream .COMP. 'ammonia'
```

Note that the prefix and suffix of the component may be omitted.

- `.ATTRIBUTE.` - The `.ATTRIBUTE.` operator returns the value of a slot attribute as a string, and is used as follows:

```
slot .ATTRIBUTE. attribute
```

5.5 Example Program

To illustrate the use of the structures and subroutines of the API, an example program is described below with numerous comments.

```
1|program example  
2|  
3|use process_objects  
4|implicit none  
5|  
6|type (component) :: comp, comp_out  
7|character*200 :: creator, filepath, error, url_path, aspen_name  
8|
```

At the beginning of any program using the API, the statement `use process_objects` must appear (on line 3 above). This makes available both the Fortran subroutines and structures to the program. Two component structures have been declared on line 6, together with various character strings on line 7.

```
9|!set the creator  
10|creator = 'ross@chemeng.ed.ac.uk'  
11|  
12|call www_setup_api(creator, filepath, url_path, error)
```

```
13|call check_error(error)
14|
```

At line 12 the first API subroutine call is made. This subroutine sets up the user's workspace on the server, and assigns a session id. It is equivalent to initially entering the system via the WWW interface, and performs many of the tasks described in section 3.3.1. A directory is created, if required, on the WWW server for the user's workspace, and this is populated with several component and mixture objects. Note that it is not possible here to supply a password to restrict access to this directory.

The `check_error` subroutine, called on line 13, appears at the end of the program, and is used to trap any errors generated by calls to API subroutines.

```
15|!create component structure
16|call www_initialise(comp)
17|
18|!set slots for creator and filepath
19|call www_set_slot(comp, 'creator', 'value', creator, error)
20|call check_error(error)
21|call www_set_slot(comp, 'filepath', 'value', filepath, error)
22|call check_error(error)
23|
```

One of the component structures declared at the beginning of the program is initialised at line 16. What this means is that the slots present in the HTML version of a component object, both data and logistical, are added to the relevant arrays of the Fortran structure. All attributes for these slots, apart from the slot name, are set to null strings.

After the structure has been initialised, values for the creator and filepath variables are entered using the `www_set_slot` subroutine.

```
24|!add the component Name
```

```
25|call www_set_slot(comp, 'Name', 'value', 'benzene', error)
26|call check_error(error)
27|call www_set_slot(comp, 'Name', 'status', 'entered by user', error)
28|call check_error(error)
29|
30|!write html version of component
31|call www_write_html(filepath, comp, error)
32|call check_error(error)
33|
```

Once the structure has been initialised, the component name is added to the structure, and the slot status set. Then an HTML version of the structure is written to the user's server workspace. There is no difference between an HTML object created in this manner, and one created using the manual object creation section of the model generation manager (see section 3.3.1).

```
34|!apply method to obtain properties from databank
35|call www_apply_method('databank', comp, comp_out, error)
36|call check_error(error)
37|
```

Next a method is applied to the HTML version of the object stored in the structure `comp`. This method, indicated by the code "databank", gets properties for the component from a databank. This is used to create a new HTML object with the data from the databank. The data of this new HTML object are returned to the program and used to create a new structure, `comp_out`.

```
38|!extract the aspen name
39|aspen_name = comp_out .SLOT. 'Aspen_name' .ATTRIBUTE. 'value'
40|print *, 'ASPEN name is: ', trim(aspen_name)
41|
```

The overloaded operators are used to access data fields of the structure. Here the ASPEN name of the component is obtained and printed to screen.

```
42|!write xml file to see results
43|call www_write_xml(comp_out,'./data.xml')
44|
45|end program example
46|
```

Finally the contents of the structure are examined by printing out a local file, containing the data in XML format.

```
47|subroutine check_error(error)
48|  !this routine checks an error message and stops the program if an
49|  !error has occurred
50|
51|  implicit none
52|  character*(*), intent(in) :: error
53|
54|  if(trim(adjustl(error)) /= '') then
55|    print *, 'error is: ', trim(error)
56|    stop
57|  end if
58|
59|end subroutine check_error
```

The subroutine `check_error` is used to check for any errors generated by the API subroutines. If an error message has been generated, then it is printed to screen, and the program execution stops.

5.6 Summary

This chapter has described the implementation and operation of an API to both create and manipulate objects on a WWW server, via a Fortran program. The API consists of several structures representing the standard objects, and subroutines to manipulate these structures and the objects. Both the structures and the subroutines have been briefly described here. To illustrate their use, an example program has been listed.

Chapter 6

Case Studies

The previous chapters of this thesis have described a system intended to facilitate process model construction, and to chronicle process model building, using WWW technology. This chapter describes four case studies, making use of this system. The first case study demonstrates how a simple model of a plant manufacturing hydrogen cyanide is constructed. Once constructed an ASPEN model is generated from the entered data. The starting point for the second case study is an ASPEN model. The data from this model are used to generate a spreadsheet model. The third case study illustrates the manipulation of an ASPEN model via a Fortran program using this system. The data are extracted from the ASPEN model, and used to generate several new ASPEN models, each with slight differences. From the simulated models, data are extracted and tabulated. The final case study shows how a user interface may be provided for an external program using this system. The external program is one which models a hydrogen cyanide reactor. A new object class is defined for this reactor, with additional slots and methods.

6.1 The Creation of a Topology Object and an ASPEN model

This section will describe the construction of a topology object, and associated objects, required to model a greatly simplified plant producing hydrogen cyanide (HCN). Pa-

parameter entry for individual objects is also described. The section finishes by examining the creation of an ASPEN model for the data entered.

6.1.1 Plant Description

A simple plant to produce hydrogen cyanide is shown in figure 6.1. It takes as feeds ammonia (NH_3) and methane (CH_4) and produces the hydrogen cyanide product and a hydrogen (H_2) byproduct.

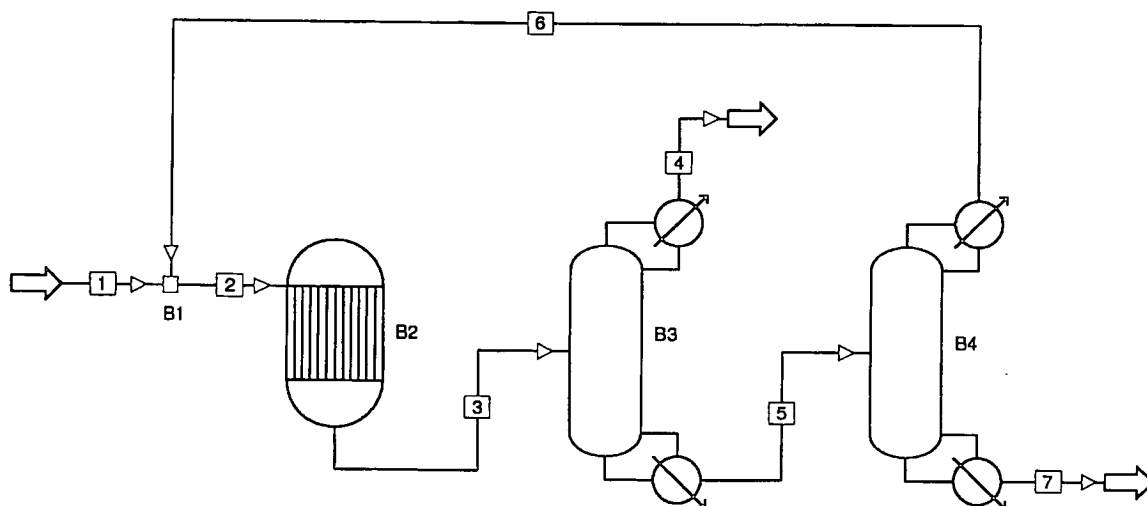
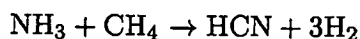


Figure 6.1: Diagram of a simple hydrogen cyanide plant

There are four main unit operations. The mixer, B1, mixes the ammonia and methane feed with an ammonia recycle stream, before being fed to the reactor, B2. The hydrogen cyanide is created by the following reaction:



The reactor products are fed to the column B3, where the hydrogen is removed. As it is difficult to separate hydrogen from methane, both of these components are removed here in stream 4. The remaining components are then passed to column B4, where the ammonia is removed and recycled, leaving the hydrogen cyanide product, stream 7.

For the purposes of simplification, no side reactions are considered, and it is assumed that distillation will be used to perform both of the separations.

6.1.2 Topology Object Construction

Before creating a topology object, it is necessary to create a mixture object containing the component objects present in the topology. This is accomplished by pressing the “add more components” button, in the “get physical properties” section of the model generation manager. From the resultant list, the four components may be selected and the “Get Component Data” button pressed. This will display a page asking the user to enter the structural formulae for each of the selected components. The formulae may be either entered manually, or built up using the molecular typewriter. The structural formula is used to estimate component data. Once these have been entered, a mixture object is created containing the components selected.

There are two methods of describing the topology: either by using a natural language description which is parsed, or by selecting an approximate structure and modifying it as required. Both methods will be described here.

Natural Language Description

The natural language description of the process is simply the process described using normal English. Certain key words are extracted and used to create the topology. This is discussed in detail in section 4.7.1.

The HCN plant shown in figure 6.1 may be described by the following natural language description:

Streams 1 and 6 are mixed, and then reacted. The products are then separated to give 4 and 5, which is separated to give 6 and the HCN product.

After parsing, this gives the following topology:

feed

```

1
mix      B1
6 1      2
reac     B2
2        3
sepa     B3
3        4 5
sepa     B4
5        6 7
end
product
4
product
7

```

Approximate Topology

As well as entering a natural language description, the topology may also be entered by selecting an approximate topology, and altering it as required. From the list of available topologies, the one closest to the HCN plant is that of one recycle with no purge, shown in figure 6.2.

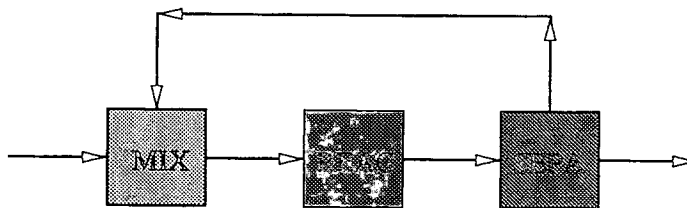


Figure 6.2: A topology with one recycle stream with no purge

This structure has the following topology:

```
#this is for a one recycle system
```

```
#if more than one feed is present then it may be added
```

```
#together with a mixture
```

```
feed
```

```
1
```

```
mix      B1
```

```
1 5      2
```

```
reac     B2
```

```
2        3
```

```

#the bottoms stream is recycled here, to recycle the top
#then swap 4 and 5 below
sepa    B3
3      4 5
end
product
4

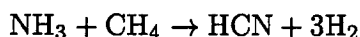
```

This topology is presented to the user and may be altered to be equivalent to that given previously.

Once the user is satisfied with the topology, the topology object, and associated stream, reaction and unit objects are created.

6.1.3 Parameter Entry

A topology object has now been created, together with component, mixture, reaction, stream and unit objects. The next task is to add data to these objects where required. Each of the unit objects are created with slots considered appropriate to their purpose, e.g the unit object modelling the reactor B2 has slots for extent of reaction, conversion, key component, reaction, temperature and pressure. A reaction object has been created, containing the components present in the mixture object. It is necessary to add the reaction coefficients for the reaction. These are the coefficients of the components in the reaction, negative for reactants, positive for products. The reaction coefficients for the reaction:



are given in table 6.1.

After entering these coefficients into the reaction object, it is necessary to apply the method “update object to register manual changes”, to the reaction object. This will create a new object containing the entered data. However, the reaction slot in the

Component	Reaction Coefficient
NH ₃	-1
CH ₄	-1
HCN	1
H ₂	3

Table 6.1: Reaction coefficients for the reaction between NH₃ and CH₄

unit object still links to the old reaction object. This may be altered by applying the method “use latest version of referenced objects”, to the unit object. This will create a new unit object, whose reaction slot now links to the updated reaction object.

The remaining parameters have to be entered to the unit object representing the reactor B2. They may either be entered manually, or a natural language description can be given which is parsed and the data extracted. A suitable description here would be:

B2 carries out reaction at pressure 10 bar, with conversion of ammonia of 90%.

Applying the method “Update object to register manual changes”, parses the description, and extracts the required data. If no data may be extracted for a slot that is considered essential, then a default value is used. This is discussed in greater detail in appendix A.

Similarly, parameters are required for the two separators: B3 and B4. Suitable descriptions for the two units are given below.

B3:

B3 separates light key methane from heavy key ammonia, with light key recovery of 99.9%, and heavy key recovery of 0.01%.

B4:

B4 separates light key ammonia from heavy key hydrogen_cyanide, with light key recovery of 99.9%, and heavy key recovery of 0.01%.

Note that in all of these descriptions, the components may be referred to without the surrounding numbers of the object names, e.g. hydrogen_cyanide instead of 101_26_hydrogen_cyanide_1_01.html.

The final task that needs to be accomplished is to add the feed stream data. The stream object representing the feed stream was created when the natural language description was parsed. All that is required is for the component flows to be entered, and the stream temperature and pressure. After the data have been entered, the method "update object to register manual changes", is applied, which creates a new stream object with the changes.

Once the new unit objects have been created with the parameter data added, the method "use latest version of objects", may be applied to the topology object, which will create a new topology object, referencing the new unit objects and the feed stream.

6.1.4 Model Creation

An ASPEN model of the process may be created by applying the method "create an ASPEN model". The ASPEN model is listed in appendix I.1. The data have been extracted from the relevant objects and used to build the model. One point to note is the name of the components. The names used to refer to the components in the ASPEN model must be unique for the first eight characters. The component names are generated by taking the first eight characters of the component ASPEN name. If this does not generate a unique name, then the component ASPEN name is prefixed with a, b, etc. and the procedure repeated until a unique name is generated. This accounts for the fact that hydrogen cyanide has the model name hydrogen, but that hydrogen has the model name ahydroge.

6.1.5 Discussion

This case study demonstrates how a topology object, together with ancillary objects, may be created and used to model a process. The hydrogen cyanide plant used here as an example is a very simple process, but a more complicated process would only involve an increase in the size of the problem, not the complexity.

Both of the methods for entering a topology from scratch have been used in this example. Each of the natural language description and the approximate topology methods of describing the plant structure have advantages and disadvantages.

The approximate topology provides a framework whereby the user may add in the remaining details of the structure. The set of approximate topologies available for the user to select from match a large number of processes. However, it can be somewhat restrictive. The generating of a topology from the natural language description gives the user more freedom, but can be more prone to errors, in that the description must be written in a particular manner. However, after the description is parsed, the user has the option to edit the resultant topology to correspond with what was intended.

As well as describing the topology via a natural language description, the unit parameters may also be described in this manner. This provides an alternative to entering the data in the form manually. The parsing also provides default values for any quantities not specified but considered necessary.

6.2 The Creation of a Spreadsheet Model from an ASPEN Model

This section will describe the creation of a spreadsheet model of a process from an ASPEN model file. The ASPEN model is one that describes the manufacture of cyclohexane from benzene. First of all a topology object, and associated component, reaction, stream and unit objects are created using data extracted from the ASPEN file. From these objects a spreadsheet model of the process is then constructed.

6.2.1 Cyclohexane Manufacture

Cyclohexane (C_6H_{12}) is manufactured industrially by the hydrogenation of benzene (C_6H_6). The reaction is:

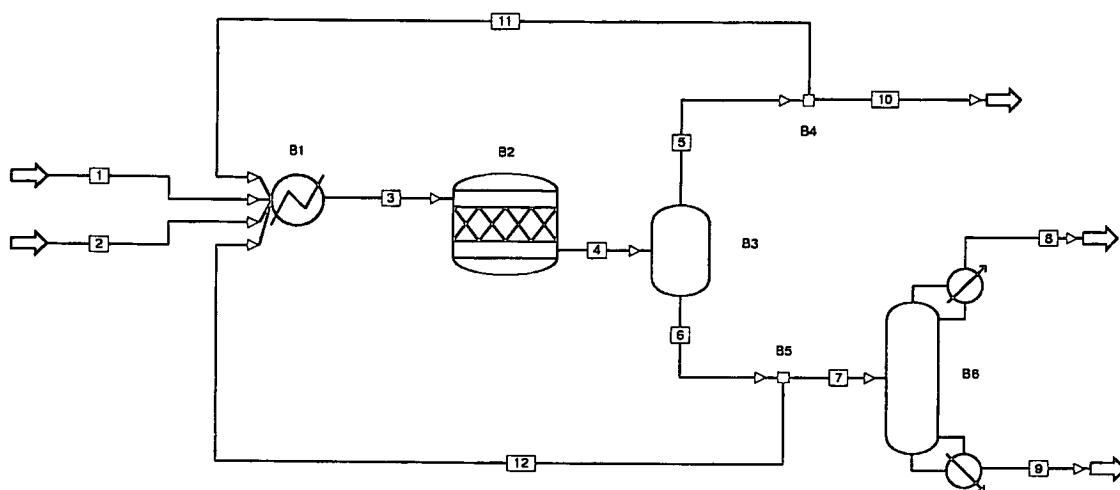
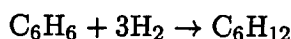


Figure 6.3: Aspen model of a cyclohexane plant

A diagram illustrating a cyclohexane plant is given in figure 6.3. This shows an ASPEN model of the plant. The input file for the model is given in appendix I.2.1, and is based on an example model given in the ASPEN documentation [2]. The hydrogen feed (stream 1) and the benzene feed (stream 2) are mixed with the recycle streams, before being heated in B1, and fed to the reactor, B2. The reactor products are then flashed in B3. The liquid products of the flash, mainly unreacted benzene and cyclohexane, are split with one fraction being recycled back to the reactor. The other fraction is distilled to recover the pure cyclohexane product (stream 9).

The vapour products of the flash are also recycled back to the reactor. The hydrogen feed to the reactor contains some methane and nitrogen impurities. To prevent a build up of these impurities occurring, a small purge is taken from this recycle stream.

A fuller description of cyclohexane manufacture may be found in the Kirk-Othmer Encyclopedia of Chemical Technology [29].

6.2.2 Topology and Associated Objects

A topology object may be created from an ASPEN model from the “Enter Data” section of the “Model Generation Manager” (see section 3.3.1). The parsing of the model is discussed in detail in section 4.7.13. Briefly, the ASPEN model is uploaded to the www server and parsed. The component names are extracted and used to create component objects, which are then used to create a mixture object. Objects are also created for the streams and units of the topology. A reaction object is created for each reaction present. These objects contain all of the relevant data that may be extracted from the ASPEN model. Finally a topology object is created to hold the stream and unit objects, and to show their connectivity. The actual topology for the cyclohexane plant is given in appendix I.2.2.

6.2.3 Spreadsheet Model

Once the topology object has been created, a spreadsheet model may be generated. The mechanics behind the model creation are discussed in section 4.7.9. Briefly the topology is used to generate equations, which are embedded within the spreadsheet. Data are extracted from the objects comprising the topology object to provide parameters for the equations, e.g. feed molar flows from feed stream objects. As would be expected, the stream flows generated by the spreadsheet are identical (to four significant figures) to the equivalent streams flows on the ASPEN model.

The actual spreadsheet model may be found in appendix I.2.4, with the full ASPEN solutions in appendix I.2.3.

6.2.4 Discussion

There are two parts to generating a spreadsheet model from an ASPEN model:

1. Extracting the data from the ASPEN model.

2. Creating the spreadsheet model from the extracted data.

When the data are extracted from the ASPEN model, it is used to create objects for the various streams and units that comprise the topology. Also created at this point are any component objects. Component objects are only created if there are no objects representing that component already created. If the spreadsheet model in appendix I.2.4 is examined, it can be seen that in the data area, hydrogen, methane and benzene have got all their properties calculated, whereas only some properties are listed for nitrogen and cyclohexane. This is due to the fact that there were already objects created for the first three components, but new objects had to be created to represent the remaining two components. The data extracted from the ASPEN model to create these new component objects are: ASPEN name, component name, the molecular weight and the atmospheric boiling point.

The boiling points of the components are used to order the components in the spreadsheet in approximate order of decreasing volatility. This is important as it enables unknown component recoveries to be estimated. Component recoveries are normally calculated by dividing the molar flow of a component in an output stream by the component molar input flow. However if there is no input or output flow then this calculation may not be performed. Instead it is estimated as described below.

First of all the calculated recoveries are examined to see if they are increasing or decreasing with increasing boiling point. If increasing, then a default value for unknown recoveries of zero is set, and vice versa. Then the ordered list of components is examined (the most volatile first), and any unknown recoveries set to the default recovery. The default recovery remains at its initial value, until a calculated recovery is encountered that is more than 0.5 different from the default recovery, whereupon the default recovery changes to be either zero or one, dependent on what it was previously. For example if the default recovery was initially zero and a recovery greater than 0.5 was encountered, then the default recovery for unknown values would then be one for the remainder of the component list.

Table 6.2: Stream flows from the two cyclohexane plant models at a reduced benzene conversion of 98%

Component	Reactor Inlet Stream 3		Vapour Flash Stream 5		Cyclohexane Product Stream 9	
	Aspen	Spreadsheet	Aspen	Spreadsheet	Aspen	Spreadsheet
Hydrogen	172.7	171.7	39.3	38.0	-	-
Nitrogen	7.5	6.9	7.3	6.7	-	-
Methane	24.4	21.1	23.0	19.4	-	-
Benzene	45.2	45.2	0.0	0.0	0.6	0.6
Cyclohexane	20.2	19.8	1.4	0.9	43.9	44.3

The spreadsheet is solved using Gaussian elimination with the Gaussian elimination code being embedded in the spreadsheet. To stop the occurrence of zero pivots when solving the spreadsheet equations using Gaussian elimination, component recoveries may not be exactly zero or one. Instead all recoveries are constrained to be greater than zero and less than one. This accounts for the small discrepancy in the spreadsheet and aspen models, listed in appendices I.2.4 and I.2.3.

The component recoveries for separators in the spreadsheet model are only accurate at the flowrates specified in the ASPEN model. However for small alterations in the flowrates, the recoveries are reasonably accurate. The advantages of using a spreadsheet model are that as all code necessary to solve the model is embedded into the spreadsheet, the spreadsheet may be transferred to another computer. Furthermore it solves in a fraction of the time taken to solve the ASPEN model. Thus for small changes in the model, it can be useful to investigate the changes by using the spreadsheet model. For example, table 6.2 shows the component flowrates for both the spreadsheet model and the ASPEN model if the conversion of the benzene in the reactor were reduced from 99.8% to 98%.

The largest absolute error in the spreadsheet flowrate is that of the methane flow in stream 5, the vapour flash. It is 3.6 kmol hr^{-1} , or as a relative error, 16%. The relative error in the cyclohexane flowrate in stream 5 is larger, due to the low flowrate of cyclohexane.

Obviously these errors are too large for detailed design work, but the spreadsheet illustrates the general effect of lowering the conversion.

6.3 The Manipulation of ASPEN from a Fortran Program

This section will describe a Fortran program which uses the API to manipulate an ASPEN model of a rigorous distillation column, so that it is simulated with three different reflux ratios, and at three different pressure levels.

The distillation column is designed to separate an equimolar feed of benzene and toluene. It is modelled in ASPEN using the rigorous distillation column model, RADFRAC. The ASPEN model may be found in appendix I.3.1. It is set up initially with a reflux ratio of 3, and operating at atmospheric pressure.

6.3.1 Implementation

The program manipulating the ASPEN file may be found in appendix I.3.2. The program first of all sets up the API using the `www_setup_api` subroutine.

Next the initial topology object, together with the associated component, mixture, stream and unit objects, are created by parsing the given ASPEN model. Then for each combination of the pressure and reflux ratio, the data are added to the unit, and an ASPEN model is created and simulated. Results are then extracted from the simulated model, and the component molar flowrates for the tops product stream is printed to an output file.

There are several subroutines contained in the program, which perform many of the tasks. These are described below.

- `check_error` - This subroutine checks error messages generated by the API sub-

- routines. If an error message is encountered, then it stops the program execution.
- `create_initial_topology` - The initial topology object, and other objects are created by this subroutine. First of all a Fortran topology structure is created, and the `filepath` and `creator` variables added to it. This structure is then written to an HTML topology object. To this object, the method “get data from an ASPEN model”, is applied, which extracts data from the given ASPEN file, and uses it to create the topology object.
 - `add_details` - The reflux ratio and pressure are added to the unit object using this subroutine. This is achieved by first of all identifying the sole unit object in the topology object, and creating a Fortran structure from this unit object. The reflux ratio and pressure are added to the structure, and are then saved to the new object by applying the method “update the object to register manual changes”. This creates a new unit object with the updated data. However the topology object still links to the old unit object, so to rectify this the method “use latest versions of objects”, is applied to the topology object.
 - `simulate_model` - This subroutine creates an ASPEN model from the topology object, and then downloads the model to the local filespace. The model is simulated, after which the input file and summary files are moved to a directory accessible by the www. The simulated data are returned to the topology object by applying the method “get data from an aspen model”, giving the URL of the input file as a method parameter.
 - `extract_details` - The component molar flowrates of the tops product stream of the unit are extracted using this subroutine. This is accomplished by identifying the tops stream object, and creating a Fortran stream structure from it. The two flowrates are then obtained from this structure.

The output of the program is displayed in appendix I.3.3.

6.3.2 Discussion

By calling the subroutines of the API, the user is only performing the same tasks as would be carried out by pressing buttons in the HTML objects. This is best illustrated by examining the history tree of topology creation if the program described above is run. The history tree is shown in figure 6.4.

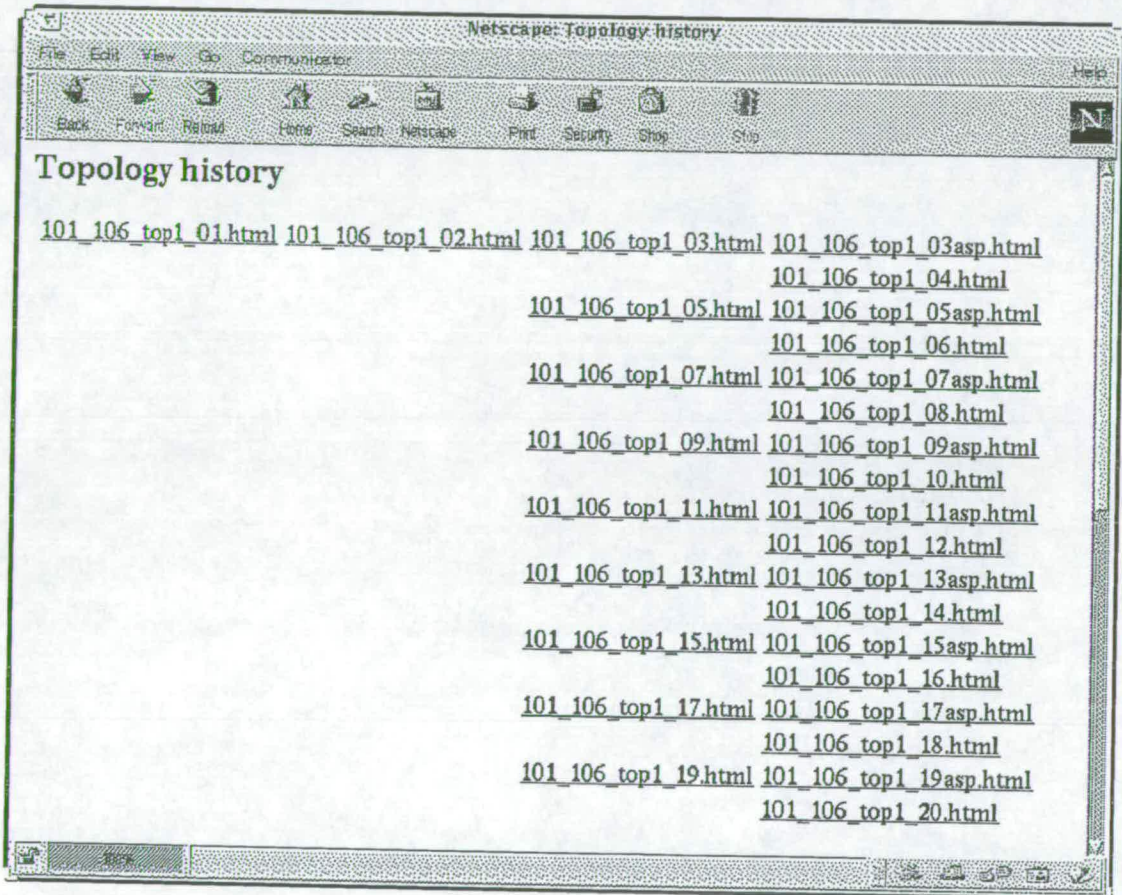


Figure 6.4: History of topology creation for API program

The first topology object in the tree, `101_106_top1_01.html`, was created when the initial topology structure was written to an HTML object. The child of this object, `101_106_top1_02.html`, was subsequently created by applying the method which parsed the ASPEN model file. There are nine children for this object, which represent the nine different combinations of the pressure and reflux ratio. Each of these children have in turn two children of their own: a wrapper around the ASPEN model, and a topology object containing the results of the ASPEN model simulation.

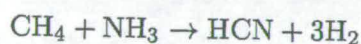
6.4 A Fortran Program with a WWW Interface

This section will demonstrate how the system described in the previous chapters may be used to provide a user interface to an external program. The Fortran program used here is one that models the reactions taking place in the production of hydrogen cyanide(HCN) from ammonia(NH₃) and methane(CH₄), using the Degussa process.

6.4.1 Reactor Description

The reactor type used in the Degussa process is a tube wall reactor, which consists of catalyst deposited on the surface of ceramic tubes. These tubes are used to transfer heat to the reactor. Several tubes are used in parallel to increase the surface area of the reactor.

The reactor model deals with only two reactions:



The rate expression data for these reactions are given in Waletzko and Schmidt [53].

The formation of hydrogen cyanide is highly endothermic, and energy must be transferred to the reactor to maintain the reactor temperature.

Program Details

The Fortran program simulating the reactor model takes as arguments: the partial pressures in bar of all five components, the total molar flow to the reactor in kmol/hr, the tube diameter in metres, the tube length in metres, the number of tubes present, and finally the temperature of the reactor in Kelvin. For the purposes of this model, it is assumed that the reactor is maintained at a constant temperature throughout.

The output of the program consists of the molar flows of each of the five components.

6.4.2 Custom Object Class

The most convenient, and sensible, method of including an executable program within the system is to encapsulate it in a custom method. This involves creating a new class of object. As the reactor program requires several arguments, it is sensible to include these arguments as data fields in the new object class.

A new object class is created by first of all applying the selection option “Create a new custom class” (see section 3.4.5). This displays to the user several selection lists using which the user must state how many new slots (data and logistical) and methods (object methods, selection options and history options) are to be defined in the new class. Here there are five new data slots, and one object method to be defined in the new class.

Once the number of slots and methods have been defined, the user is then presented with textboxes in which the data may be entered. Before entering this data, a class name must be given, together with the object class from which this object will inherit from. The class name must be a single word, and not the name of an existing class. Every new object class must inherit from an existing class. If none of the standard classes are suitable, then the new class may inherit from the `Base_object` class. This will provide the new class with the basic logistical slots, and the standard object methods, other selection options and history options. For this example, calling the class `Reactor` is sensible. Since a reactor is a type of unit, inheriting from the `Unit` class also makes sense.

The data slots are entered by typing their name, identifying them as single slots, and selecting the option which says they have the standard name, value, description and status attributes.

The object method is slightly more complicated. It requires three parameters. The

first is a single word by which the method will be referred to if using the API. Next comes a sentence description which is the name by which the method will be known in the object. Finally the URL of a CGI script is required which takes as parameters an XML file containing the object data and the URL of the directory where this file resides. This script should return a new XML data file, modified by the external code. This is an external script, executed on a file system where the user may run CGI scripts.

The full class file for the class `Reactor` is given in appendix I.4.1, whilst the CGI script encapsulating the Fortran code is described below.

CGI Script

The CGI script is passed two parameters: the URL of an XML file containing the data of the object to which the method is applied, and the URL of the directory in which this file resides. It is up to the script to extract the necessary data required for the Fortran program, to run the Fortran program, and to write an XML file containing the modified data.

The code for the CGI script is given in appendix I.4.2. The script first of all downloads the XML reactor file, and extracts the name of the stream object from this file. The XML file of the stream object is also downloaded. Next a temporary file is created to contain the Fortran program arguments. To this file are added the feed stream data, and the reactor data. The program is then run and the output stream data obtained, which are used to create an output stream XML file. Finally a reactor XML file is written, and the URL of the reactor XML file is returned from the CGI script.

The script contains several subroutines, which are detailed below.

- `add_stream_data` - This subroutine extracts the component flows for each of the components, works out their partial pressures, and adds these to the program data file together with the total molar flow.
- `add_reactor_data` - This subroutine adds reactor data to the program data file.

- `add_mole_flow` - The output stream XML file is created by this subroutine, adding the molar flows of each of the components. A check is made first of all to see if the component object is present in the feed stream. If not, then an XML file for the component is generated.
- `create_xml_reactor_file` - The new reactor XML file is generated by this subroutine. The product stream name is added to the old data, and some logistical data altered.
- `return_error` - This subroutine returns any error messages. If the script runs correctly, then it should return a URL. If a URL is not returned, then it is assumed that an error message has been generated, and this is displayed to the user.
- `get_slot_attributes` - The slot attributes for a given slot are extracted using this subroutine.
- `extract_stream_data` - This subroutine extracts the names of the component objects from the input stream XML file.
- `write_xml_intro` - The general introduction to an XML file is written by this subroutine.
- `get_flow` - The molar component flows are extracted from an XML file by this subroutine and returned.

6.4.3 Method Application

Once the `Reactor` class has been defined, the next task is to create a reactor object. This may be done by displaying the model generation manager, and going to the “create objects manually” section. This section will display a selection list from which the `Reactor` class may be chosen, and a reactor object created.

The reactor object created here has all its slots empty. Before the method “model an HCN reactor”, may be applied, the data slots must be filled, and a feed stream added. The data for the data slots may be added manually and the method “update object to register manual changes”, applied to save the changes in a new object. Adding a feed stream is more complicated.

First of all a new stream object must be created. This may also be created in the “create objects manually” section of the model generation manager. However, when creating a stream in this manner, a mixture object must be selected which contains the stream components. The mixture object is created in the “physical property input section”, described in section 3.3.1.

Once the stream object has been created, the component flows for the component may be added.

Next this stream object must be linked to the reactor object. This is accomplished by applying the method “add a slot to the object” (see section 4.6.2). Entering the word “feed” in the Slot name box, the stream name in the Value box, and the name by which the stream will be referred to in the object in the Description box, adds the stream object as a feed stream to the reactor object.

Once this has been carried out, the new method may be applied to the object. As far as the user is concerned, there is no difference between applying a user defined method, and a standard method. If figure 6.5 is examined, it can be seen that the custom method “Model an HCN reactor”, appears at the end of the standard unit methods.

Upon the method application, an XML file is written containing the data of the object to which the method was applied. Also written are XML files of all objects that are linked from this object (aside from parent and children objects), either directly or indirectly via an intermediate object. Then the CGI script performing the method execution is run. This downloads several of the XML files, and uses their data, together with the data generated by the Fortran program to create several new objects.

If the script has been run correctly, then it will return the URL of an XML file of the reactor. This XML file is used to create a new object which is displayed to the user. Linked from this reactor object are two stream objects: the feed stream and the product stream. The feed stream exists as an object in the user’s server workspace, but the product stream does not. Instead an XML version of the object has been created along with the XML reactor file. The output stream XML file must be uploaded and the data

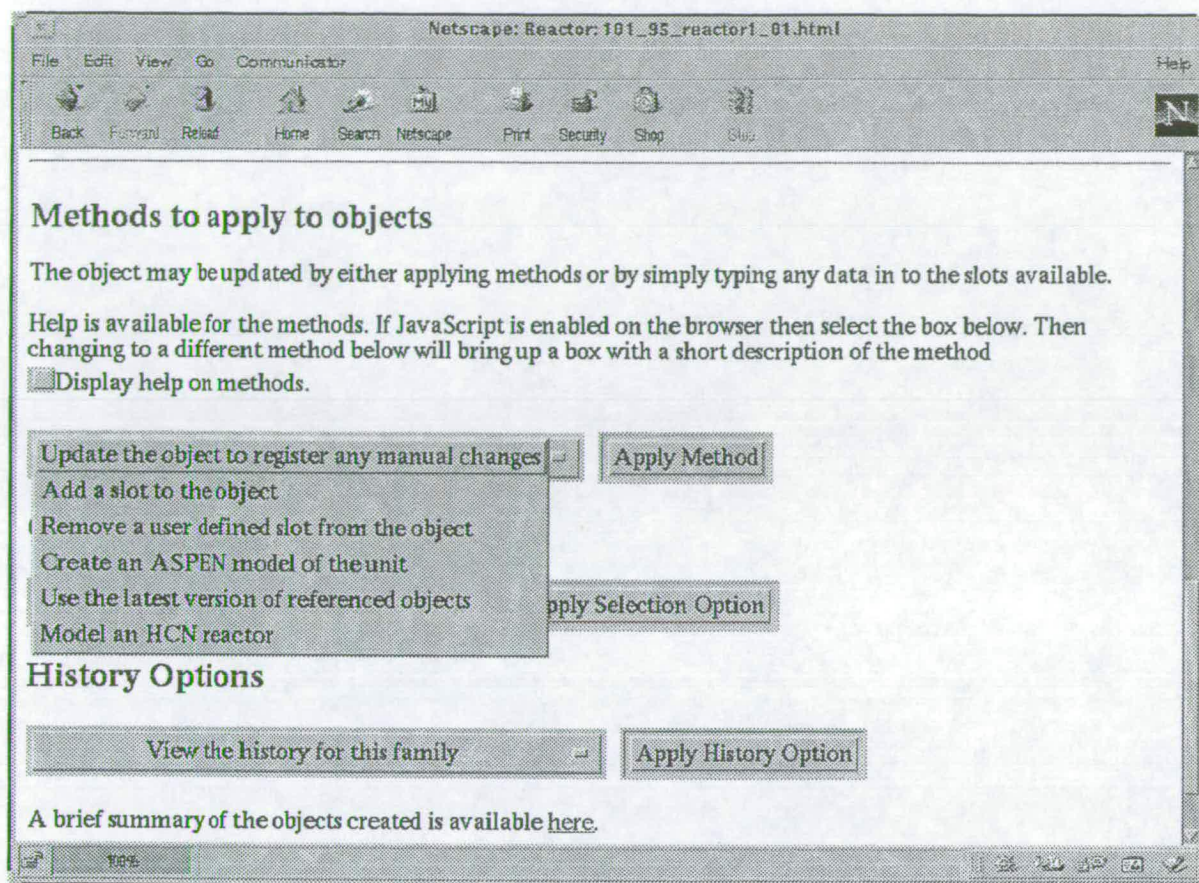


Figure 6.5: The methods of a reactor object

parsed and used to create the stream object. This stream object also contains several component objects. For those components not present in the feed stream, but created in the reactor, XML files were created. If no HTML objects exist for these components, then the XML versions will have to be uploaded, the data parsed, and new component objects created.

6.4.4 Discussion

There are two main reasons as to why the interface to the Fortran program was implemented in this manner, i.e. run via a CGI script on an external WWW server. The first of these is security. Executing a program written by someone else on the local file system can be fraught with danger. It has the opportunity to cause damage, either accidentally or maliciously. The only solutions to this are to either trust the program,

as with commercial software, to read and approve the source code, or to write the program oneself. Executing the program on an external server transfers most security risks to the user. The only concerns remaining are the fact that files are returned from executing the program. However, as these are parsed and the data extracted, the security risk is minimal.

The second reason for running the program on an external server is ease of execution. The user, or the system, does not have to compile the program for a particular operating system, have to write the program using certain languages, or rely on libraries of code being present. Instead, the user can write the program in whatever language is desired, as long as the resulting program may be executed on their own filesystem.

This application involved the creation of a new class by the user, which inherited both methods and slots from an existing object class, the unit. This provides a convenient way of both entering the required data and executing the method code. However, there are limitations to this approach. The class file is stored in the user's scratch space. This means that it will be deleted along with the rest of the user's data once the session has finished. The class would then have to be redefined before new objects of that class could be created. Also, as the class file is stored in the user's scratch space, this means that no other subsequent user is able to access it to create objects of that class.

With regard to the Fortran program simulating the reactor, the CGI script which runs this program is rather long. This is due to the fact that data had to be extracted from two different objects: the reactor object and the feed stream object. Once the Fortran program had been run and the data obtained, a new reactor XML data file, and output stream XML data file had to be created. The script also had to cope with the possibility that new component objects may be required.

6.5 Summary

This chapter has examined four different case studies of the system developed. These were:

- creating a topology object, and hence ASPEN model, from scratch
- creating a spreadsheet model from an ASPEN model
- manipulating an ASPEN model from a Fortran program
- providing a user interface for a Fortran program

The next chapter contains a discussion of the system in general.

Chapter 7

Discussion

This chapter discusses the main system features. First of all the criteria given in chapter 2 for a process engineering information management system are examined. Some criteria are met by default by using WWW technology. Others are only partially met, or are not met at all. The system is examined to show how those criteria not totally met have been fulfilled. Next the use of WWW technology in this system is examined. The advantages and disadvantages of using HTML, browsers and CGI scripts are assessed. After this comes a review of the API. The chapter concludes by reviewing the different methods of model construction using this system, and detailing the various simulation models that may be generated.

7.1 Process Engineering Information Management System Criteria

In chapter 2, a list of criteria for process engineering information management systems were given. It was noted that by basing a system around the WWW, several of these criteria are met in full, some are only partly met, and others are not met at all. The criteria not completely fulfilled by using the WWW are:

- **Efficiency** - If the overheads are large, then users will be reluctant to use the system. This applies to the time to access the system, to run simple tasks, and

to learn the system.

- **Persistent Data Storage** - It must be possible to create persistent data structures and to be able to access them over a period of time.
- **User Extensible** - The system must be easily extensible by the user, and not require computer experts.
- **Application Integration** - It must be possible to integrate external programs into the system.
- **Audit Trail Maintenance** - There needs to be a record maintained of user interaction with the system.

Each of these criteria are discussed in the following sections to show how they have been met by the system, and how well the WWW copes in meeting them.

7.1.1 Efficiency

There are three facets to the efficiency criterion. The system should be quick to access, it should be efficient at running simple tasks, and a large time should not have to be invested in learning the system.

Access Time

The time taken to access the system is governed by the quality of the user's connection to the WWW. Dubbed by some as the "world wide wait", on occasions access to files across the WWW can be quite slow. This can be due to network congestion, or to a hardware or software fault either on the server or the network. Network and server faults are outwith the system's control. Network congestion may be alleviated by reducing both the number of network calls, and the volume of network traffic. Each time a method is applied to an object, there has to be communication between the client and the server. The object data are passed from the client to the server and, providing the method application was successful, a new object is created on the server and returned to the client. To reduce the volume of network traffic, all object files are

kept to be deliberately small, and there are very few graphics files, which tend to be the largest files commonly transferred.

The number of network calls may be reduced by trapping errors at the client. If a method application is not successful, then all that the client - server communication has achieved is to inform the user that an error has occurred. In several cases, client-side programming is used to try and identify any errors before applying the method, e.g. when adding a new slot to an object, a JavaScript function is used to ensure that a name has been given for the new slot.

Efficient at Running Simple Tasks

In discussing the previous facet, it has been mentioned that the time taken to access the system depends on the WWW connection. This also affects the time taken to run simple tasks. The other factor which may affect running times is if several users apply methods in a short period of time. As virtually all processing is carried out on the server, this could cause congestion and slow down program execution. The solution to both of these problems is to transfer more processing from the server to the client.

In most cases, the client is used as a dumb terminal. Little or no use is made of its potentially large processing power, which could be utilised using Java applets. Due to restrictions on applets not being able to write files except on the server from which they were downloaded, it is not possible to replace all server-side programming, with client-side programming. However, it should be possible to use more client-side programming to reduce both the network traffic, and the processing burden on the server. This could involve error checking to ensure that all data required for a method has been entered, or pre-processing object data to only send to the server the data that are required, rather than all the object data.

Time to Learn System

The third facet to the efficiency criterion is that of the time taken to learn the system. There are two attributes to this: the time taken to learn the interface, and understanding of the concepts behind the system. Most people are familiar with WWW browsers and are comfortable with the various aspects, e.g. clicking on a hyperlink to load a new page, entering data in text boxes. With regard to the concepts behind the system, by and large process engineers are familiar with terms such as components, mixtures, streams, and their various properties.

7.1.2 Persistent Data Storage

Conceptually, data are stored and manipulated using objects in this system. The object oriented paradigm provides a convenient mechanism for encapsulating data in a structured framework. Object classes have been defined for common process engineering entities such as components, mixtures, streams, etc. Within these object classes, data slots have been declared for class properties, e.g. components slots in the mixture class.

Once instantiated, objects are stored in HTML files. The main advantage of using HTML is that it is an open standard, used worldwide, and controlled by a non-profit making organisation, W3C [57], rather than by a single company. Also no custom viewer needs to be written to access the file. Any standard browser may be used. However, it does mean that before any object data are used, the HTML file must be parsed and the data extracted. This slows down method execution and increases the processing load on the server.

For each user, a directory is created on the WWW server. All of the object files generated by the user are stored in this directory. After a twenty-four hour period during which no new objects are created, this directory is removed. The user may save a record of their work by applying the method which compresses all created objects and emails them to the user. This file can be uncompressed, and objects viewed. Methods can

still be applied to these objects, although first of all the object must be uploaded to the www server. Thus persistent data storage has been achieved.

One outcome of this approach is that the user may place the objects such that they are accessible via the www. This may be thought of as analogous to the concept of “publication”, as defined by n -dim.

7.1.3 User Extensible

There are two parts to this criterion: extending existing objects, and creating new object classes. Objects are created with slots considered appropriate to their function, e.g. component objects have a slot for the component molecular weight. This list of slots may be extended by applying the method that adds new slots, which extends only the current object. It does not alter any past, present or future instantiations of the object class, other than the object to which the method was applied.

New object classes may be created by inheriting slots and methods from existing object classes, and defining new ones. The addition of new slots is relatively straightforward, in that the slot name and type are specified via an input form. However the adding of new methods is more complex. For reasons described in section 3.4.5, methods must be executed via a CGI script. This proves to be the approach by which the widest range of programs may be supplied as new methods. The writing of the actual method code will not be discussed here as this is not what this criterion is concerned with. This criterion is that a the system must be easily extensible without need of a computer expert.

New slots may be defined for existing objects, and new object classes created with additional slots without writing any program code. The only situation where a program is necessary is the addition of new methods, where a CGI script is required. To be useful, the minimum that this script may do is to download the XML file of the object to which the method is applied, extract the relevant data, modify it and finally write the modified data in XML format. XML has been selected as the format to export the data as it is

becoming more prevalent as a data exchange format. There are several libraries of code available for parsing and extracting data from XML files, in the languages commonly used for CGI scripts. Thus it should not prove too difficult to extract the required data from the file.

7.1.4 Application Integration

Application integration is achieved in this system by using a common data format, and translating the input and output of external applications between this common data format and the application format. Several tools have been integrated using this method, e.g. an external simulation package, ASPEN, a custom physical property package.

The common data format used is HTML, which structurally is not an ideal format by any means. Primarily designed as a display language, it must by necessity have tags which control the layout as well as contain the data. To obtain the object data, the HTML code must be parsed and the relevant data extracted. This all takes some time and processing power. When only one object is being parsed, this is not excessive. However, where several objects are involved the time taken can be quite large, e.g. when generating history trees for all object classes, every object in the user's workspace must be parsed for the parent and child objects.

7.1.5 Audit Trail Maintenance

The audit trail is recorded in two separate locations. Firstly each object has logistical slots, which contain information such as who created the object, when, using what tool, etc. Also contained within the logistical slots are a slot for the object from which this object was created from, and a slot for any objects subsequently created from this object. These objects are hyperlinked, so it is possible to traverse the history tree of object creation using these slots. Alternatively, using the data from these two slots, history trees may be generated for the current object family, all families of the object

class, and for all objects in the user's workspace. Again the objects in these trees are displayed as hyperlinks, so the user may view any object in a browser. As well as trees, manifest lists may also be generated. These display to the user in a tabular format the object name, its creator, the method applied to create the object, the object parent and children, and finally when the object was created. The parent and children objects are hyperlinked. These lists are organised in the various object classes, and the user has the option to display the manifest list for the current object class only, or for all object classes.

Both the manifest lists and the history trees are generated from the logistical slots of the various objects at the request of the user. The second location where the audit trail is stored is the logfile. This file contains the name of every object present in the user's workspace, the method used to create it, and the time of creation. Again, the object name is hyperlinked. This file is appended to each time a new object is created.

This demonstrates that the audit trail is recorded in this system, and may be displayed to the user in a variety of different formats.

7.2 Use of WWW Technology

The previous section noted some disadvantages to using www technology. The weaknesses mentioned were that the www can be slow and is not one hundred percent reliable, and that HTML is not an ideal format for data storage. The www has other failings which impinge upon this system.

There are standards for HTML which all browsers are supposed to conform to, but this is not always the case. Browser manufacturers sometimes add their own HTML tags, which if used mean that the resultant page can only be displayed properly using that particular browser. This may be alleviated by only using the standard tags. Also the same HTML code may be rendered and displayed differently by different browsers.

Browsers do not support a reliable client-side programming environment. Some browsers

have the option to have both Java and JavaScript turned off, and other browsers do not support them at all. Also, different release versions of the same browser may support different versions of Java and JavaScript. As these languages are still developing, the older versions have fewer and more limited features. It is these versions that are supported by the majority of browsers.

However, despite these failings, using WWW technology has numerous advantages for such a system as have been described elsewhere in this thesis. The main advantage is that it provides a user interface for all but the most obscure platforms. Users are familiar with browsers, and comfortable with interacting with them. Also the client - server aspect of the WWW provides remote access to the system from any point at which a user may log on to the WWW.

Methods are applied by CGI scripts and may consist of subroutines or external code. The external code may be written in any language, as long as it can be executed on the local filesystem. This is ideally suited to the componentware approach. Instead of one monolithic program performing every task, there are numerous smaller programs, each with a tightly defined task. This makes upgrading and improving the system relatively straightforward. The disadvantage to using CGI scripts is that all processing is carried out on the server.

In assessing the strengths and weaknesses of using WWW technology, it is clear that the advantages provided by the WWW far outweigh any disadvantages.

7.2.1 Client-Server Processing Distribution

Virtually all of the processing in this system is carried out on the server. Little or no use is made of the processing power of the client. This is the result of a conscious decision made at an early stage in this project, i.e. to use the server for the majority of the processing. At that point the only practical client-side programming environments were provided by Java and JavaScript, which were not widely supported by the browsers of the day.

There have been several developments in the subsequent years. Most browsers now support Java and JavaScript, although as mentioned in the previous section the user may have them disabled. Also, the processing power of the average client has increased dramatically. Thus it would now be possible to develop a similar system, making use of the client for the majority of the processing.

7.3 API discussion

The API provides an alternative interface to the system. Instead of using a browser, the user may write a Fortran program to create and manipulate objects. Fortran was selected as the language in which to write the API to illustrate how an unfashionable language, not designed to operate with the WWW, may be used to manage the process engineering objects. The API consists of a series of Fortran structures which represent the standard process engineering objects, and subroutines which manipulate these structures. Objects are created and manipulated by taking the data from the relevant structure, and passing the data to the equivalent CGI script that would have been run if the browser interface were used. This means that the objects created using the API are identical to those created using the browser.

One major advantage to this approach is that, to use the API, only a limited number of programs need to be distributed to the user. The majority of the executable code remains on the server. The disadvantage of this is that there will still be the delay inherent in the communication between a WWW server and a client. If several calls have to be made, this can significantly slow down a program execution.

Communication between the Fortran program and the CGI scripts takes place by means of Perl scripts. These scripts take the data from the Fortran programs, and send it to the CGI scripts in the correct format. The data returned from the CGI script are then written to a format which can be easily read by the Fortran program.

The big advantage of using an API instead of a browser is that any repetitive application of methods to objects may be performed using a program, rather than having

to manually apply the methods to the objects on the browser. This is illustrated in the case study described in section 6.3, manipulating an ASPEN model using Fortran. This repeatedly carries out the same method applications with only very minor object changes at each application.

7.4 Model Construction

In this system, a process model is represented using a topology object. This stores the unit types and streams, and their connectivity. There are three methods of entering a topology: giving a natural language description of the process, selecting from a limited range of templates and modifying accordingly, and finally by supplying an ASPEN model file containing the relevant process. Each of these methods have their advantages and disadvantages.

Entering the natural language description gives the user the most freedom. The topology may be specified using everyday English, and the data extracted from this. However, an inexperienced user can easily make mistakes. The sentences must be constructed in a particular manner, otherwise the wrong topology will be created. To alleviate this, the user is presented with the parsed topology, where any alterations may be made to the parsed version.

By selecting from the range of template topologies, the user avoids these opportunities for errors. There are six templates to choose from, which cover a surprisingly large range of processes. For any processes not covered by these templates, the user may select the one closest to the process, and alter it accordingly.

The final method of entering a topology is by supplying an ASPEN model. Several different types of model may be used to create the topology: a backup file, a summary file, or an input file. One advantage of using this method, is that the user may use the graphical user interface of ASPEN, model manager, to construct the model and then transfer it across to the system. This illustrates how an existing GUI may be used to create objects. Alternatively, if an ASPEN model has already been constructed, then a

topology object may be generated from this. As well as the topology object, objects are also generated for the streams, units, components, and any reactions present in the process. However, not all of the ASPEN unit models are supported, and only some of the data in the ASPEN file are extracted.

Of the three methods, entering a natural language description gives the user the most freedom, and is potentially the most powerful. However an inexperienced user may find that entering a description such that it parses correctly is not a simple task. Selecting a template is much easier, and less prone to errors. However it is much more restricted in that the user has a limited choice of templates. Supplying an ASPEN model to create a topology is slightly different, as in most cases the ASPEN model will have been created to simulate the model. This is still a valid method, and for people who prefer using a graphical user interface, provides a means of entering a topology by using one.

Once a model has been constructed, a number of different types of simulation models may be generated from the entered data. Firstly an ASPEN model may be created, if not already in existence. This is in the input file format. However, this may be converted into the graphical backup format using the ASPEN simulation engine. Another commercial simulation model which can be generated is a HYSYS model. A more simple model is the spreadsheet model. This consists of simplified linear equations for each of the unit models, which are embedded in the spreadsheet. There are spreadsheet cells to input the unit parameters, and the component molar flows for each of the streams are displayed in a stream table. The final model is the equation object. This consists of the same simplified linear equations as used for the spreadsheet model, only this time they are made visible to the user. The equations may be added to or edited and the equation set solved.

7.5 Summary

This chapter has discussed several features of the system. First of all the list of criteria for a process engineering information management system, given in chapter 2, were

examined. It was noted that several of these were met by default by using the WWW. Each of the criteria not fully met were examined, and it was shown how the system fulfilled each of these remaining criteria. After this, the use of WWW technology in the system was assessed. It was noted that although there are several disadvantages to the WWW, the advantages provided far outweigh any drawbacks. Then the API was reviewed. Finally, model construction was assessed, detailing the methods by which a topology object may be constructed, and the various simulation models that may be generated from a topology object.

The next chapter contains a set of conclusions, and some ideas for future work.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis has described a system which makes use of the WWW to chronicle process model construction. Conceptually, the process model data are stored and manipulated in objects. The object-oriented paradigm provides a convenient framework for encapsulating data in a structured manner. User extensible object classes have been defined for common process engineering entities such as streams, components, etc. Objects may be extended by either adding slots to individual objects, or by creating new object classes, inheriting from an existing class and defining new slots and methods.

Each object consists of an HTML file and is immutable. Once created it may not be altered. Instead, if the user wishes to make a change, the object data are copied, altered, and used to create a new object. Links are maintained in the logistical slots in both objects showing the relationship between the two objects.

The logistical slots of an object contain information over and above the object data. These slots store information such as who created the object, when, and using what tool, and record the history of user interaction with the system. Using the information contained in these slots, browsable history trees of object creation may be generated. Thus if an error is discovered at a later date, it is easy for the user to retrace their

steps to the point at which the mistake was made.

Process models are represented using a topology object. Three methods of creating a topology object have been demonstrated in the case studies: giving a natural language description, selecting from a set of templates, and supplying an ASPEN model. Once created, several simulation models may be generated from a topology object. These include both commercial models (ASPEN, HYSYS), and custom built models (spreadsheet model, equation object).

As well as accessing the system through a WWW browser, a Fortran API has been written which enables the user to create and manipulate objects via a Fortran program. This enables repetitive method applications to be carried out via a program, instead of having to continually press buttons on a browser.

This thesis has shown that it is possible to create a complex system of this type using WWW technology. By using WWW technology, several benefits are provided with minimal effort. Using HTML as a data storage format means that no custom viewer needs to be written to view the data. Also, due to the ubiquity of the WWW, there will almost certainly be a browser for all but the most obscure platforms. The client-server architecture of the WWW means that a user may access the system from any point on the WWW throughout the world. Most people are familiar with the operation of browsers, and are comfortable with their various aspects, e.g. clicking on a hyperlink to load that document. Other benefits are provided with slightly more effort. Methods are applied to objects using CGI scripts, and may consist of subroutines or external code. As this is executed on the local file system, the external code may be written in any language that may be run there. Also the architecture is ideally suited to the componentware approach. Instead of having one monolithic program performing all the tasks, there are numerous smaller programs, each with a tightly defined task. This makes upgrading and improving the system extremely simple and straightforward.

8.2 Future Work

This thesis has described a system which chronicles process model construction using WWW technology. However the system has limitations which need to be addressed. As has been mentioned in previous chapters, the development of the system has been somewhat ad hoc. Decisions were made on the system structure and architecture for particular cases, which were then applied to the whole system. Indeed, in a survey of web-based applications (defined as those that require the WWW for their correct execution) Huang and Mak [28] noted that this was a feature of many of the systems they examined, i.e. that the development of applications lacked a systematic and rigorous approach.

This lack of rigour manifests itself most obviously in this system in two areas: object definitions, and the use of files as opposed to databases to store object data.

8.2.1 Object Definitions

Object definitions were derived from those used by the creators of *épée* [15]. The object set arrived at was a simplified subset, both in the number of object classes, and the list of data slots in each individual class. The object data slots were arbitrarily selected, with the main criterion being to create a much reduced data set.

The resultant set of object classes only contain the object data slots, and the methods which may be applied to the object. Other than inheritance, no information on relationships between objects are stored, i.e. no information that a particular slot may only contain a certain class of object or type of data. This may be inferred, but it is not explicitly stated in the object definition. Presently a data slot may accept any form of data (e.g. string, number, object, etc.), even when it is patently obvious that only a specific type is required, e.g. component slots in a mixture object should only be filled with component objects.

Thus there are two problems with the object definitions: the list of object classes and

their data slots, and the lack of classification of the data stored within the slots.

To rectify these problems, a data model is required, which will properly document the relationship between object classes, and properly define what may be entered to any given data slot. This data model may be stored in an XML document type definition file, which can record not only the object class and slots, but also the types of data that may be entered to the slots.

Several data models for process engineering already exist. The most rigorous is that developed for STEP. However, as has already been mentioned in the technology review chapter, this is an exhaustive definition of all information considered necessary. Hence any data model developed from this definition would be extremely complex.

Another data model is the one developed by Bayer et al. [6]. This data model is designed for the integration of existing data models for process design.

These data models, together with the *épée* object definitions need to be examined, and a simple, rigorous data model constructed for this system.

8.2.2 File Based Approach

When developing WWW applications requiring data storage and or access, Huang and Mak [27] recommend using a three tier architecture (see figure 8.1).

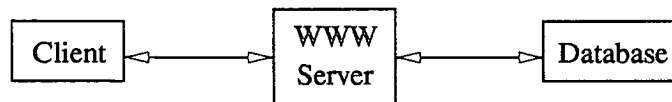


Figure 8.1: Three tier architecture for WWW applications

In the system described in this thesis, the third tier is not a database, but a set of files (one for each object), and routines to parse these files to extract data.

The decision to use a file-base approach (as opposed to using databases) to store data was taken in order that to view the data, the user would simply view a file. However,

as a result of this approach, every time object data are required, the object data file must be parsed and the data extracted.

When parsing a single object file in applying a method, the time taken to parse the file is considerably less than the time taken to contact the server, execute the method code, and return the results back to the client. However, if several objects have to be parsed for a single method application, then this time increases with the number of files parsed.

Any future version of this system should therefore use a database instead of the file-based approach currently employed. This should help reduce method execution times, especially for more complicated methods.

8.2.3 Other Problems

As well as the lack of a systematic and rigorous approach to developing the system, there are several other areas that any future work should focus on. These are discussed below.

Data Consistency

In any system consisting of related data sets, there is potential for inconsistencies. If, for example, data set A were derived from data set B, and then set B was altered without altering set A in an identical manner, then the relationship between the two data sets would be incorrect, i.e. there is a data inconsistency.

Within the system, inconsistencies can arise in a number of different places. Consider for example a reaction object with a specified reaction, i.e. component objects and reaction coefficients are entered. By applying the method "calculate reaction properties", several reaction object data slots are filled. If the user subsequently removes a component object which takes part in the entered reaction, then a new reaction object is created without this component object, but which still contains the reaction prop-

erties calculated using it. These reaction properties would then be inconsistent with the new reaction and reaction conditions. For this particular example the object data could be made consistent by simply removing the values for the reaction properties, and allowing the user to recalculate them if required.

Although this approach would work, it would remove a lot of data from the object, much of which is still consistent. Another approach has been suggest by Bogusch et al. [11]. To enforce data consistency, they suggest propagating any alterations in the data to reflect these changes. In the example above, this would mean calculating new reaction properties, which is not very complicated. However if this reaction object, were part of a larger topology object, then propagating these changes would be much more complicated. Nevertheless, some propagation of data changes should be included in a future version of the system.

Another source of inconsistencies is where the wrong sort of data are entered in data slots, e.g. an object is required but a number is entered. Having and enforcing a data model should remove this source of data inconsistencies.

Concurrency

Models are constructed and refined over time by multiple people. Any system purporting to support or document model construction should reflect this reality. However, at present this system is primarily intended to be used by a single person. Multiple people may use the system, but no record is kept of different users. It is assumed that the person who created the object is the same person that set up the server workspace at the start of the session.

To enable different users to access the system simultaneously in a coherent manner, user authentication must be present. This should identify which user created each of the objects, and control what permissions each user has, i.e. can they just view the objects, can they apply methods, or can they finish the session and remote all created objects.

As well as authenticating the users, the system should also maintain tighter control over the objects. It should optionally enable a user to “check out” an object, in much the same way as version control systems [48], to make this object read-only to other authorised users. The person who checked out this object would be able to apply methods to it, and when finished would “check in” the object to allow other users with the correct privileges to apply methods to it. This approach would cut down the number of divergent paths which may otherwise be generated from any given object.

Larger Scale Models

The models described in previous chapters of this thesis consist of less than one hundred objects. In the final stages of a design, there could well be thousands or even tens of thousands of objects in a model. These objects need to be stored and accessed in an efficient manner. The use of databases has already been discussed earlier in the chapter. Using one would improve efficiency in accessing data. However, the system efficiency would also have to be improved.

At present, virtually all processing is carried out on the server. The program execution speeds are sufficiently fast for a single, simple program. However, if several programs are running on the server concurrently, then the execution time of an individual program slows down accordingly. To speed up the system, as much processing as possible should be transferred from the server. This could be achieved by implementing the simpler methods as Java applets.

Implementing all of these recommendations should enable the system to fully support simultaneous access by multiple users in constructing and chronicling process models.

To conclude, we set out to develop a system to chronicle process model construction using www technology. This thesis has described an object-oriented system which fulfils such a role.

Appendix A

Unit Description Parsing

In the description slot of the unit object, a natural language description of the unit may be entered. If one is entered, then it is parsed and data extracted and used to both create and fill slots in the unit object. The parsing works by checking each word of the description in turn against a list of key words, which indicate the function of the unit. The key words recognised are shown in table A.1.

If any word in the description starts with a key word, then a match has been made. This means that **separa** matches all of separator, separate, separates and separated, without having to include different key words to cover for all eventualities. All matching is case insensitive.

Table A.1: Keywords recognised by parsing and indicated unit functions

Keyword	Unit Function
separa	distillation column
sharp	distillation column
split	splitter
react	reactor
flash	flash
non-sharp	flash
mix	mixer
cont	contactor
strip	stripper
absorb	absorber
heat	heater
cool	cooler
vapo	vaporiser
cond	condenser

Once a key word has been recognised, then the description is parsed again searching for information pertinent to its function. This is achieved by using a new set of key words, with different sets for each function. Each word is checked against the list of key words as described previously, i.e. if the key word is identical to the start of the word then a match has occurred. Once a match has occurred, then a flag is set indicating that the next item of a specified category is to be set to a given attribute. All other flags requiring an item of the same category are set to null. For example consider the following extract from a description of a flash unit:

... flashed at a pressure of 15.0 bar ...

The keyword "pressure", is recognised by the parser and a flag is set such that the next real number matched (15.0) will be set to the unit pressure.

However if the following extract is examined, again from a flash unit description:

... flashed at a high pressure and a temperature of 400.0 K ...

As in the previous extract, the keyword "pressure", is matched. Before a real number is matched however, the keyword "temperature" is encountered. This too sets a flag such that the next real number matched is set to the unit temperature, returning the pressure flag to null.

Other than key words, various other categories are matched in the sentence:

- **Real Numbers** - a real number consists of digits, and either a decimal point or a percentage sign, e.g. 25.1, 9%
- **Block Identifiers** - block identifiers are made up of a capital letter, followed by between one and three digits, e.g. B1, R105
- **Reaction Objects** - a reaction object must conform to the actual name of an object which exists in the user's workspace, e.g. 99_214_reaction1_03.html
- **Component Names** - a component name must be the name of a component which exists in the user's workspace. It may have the numbers prefixing and suffixing the component name removed, i.e. 98_45_methane_1_03.html or methane.
- **Stream Names** - a stream name must be the name of either a feed or product stream to the unit, given in the feed or product stream slots.

Each of the unit functions, including the list of keywords, are discussed below.

A.1 Distillation Column

A distillation column separates a single input stream into two output streams: the tops product and the bottoms product, usually of different flows and compositions. If the description is recognised as describing a distillation column, then it is parsed again to extract further data required to set up a short cut distillation model. For this it needs the heavy and light key components. The light key is the component with the highest volatility that appears in the bottoms product, and the heavy key is the component with the lowest volatility that appears in the tops product. The fractional recoveries of these components from the feed stream to the tops product stream is also checked for. Other data looked for in the description include: the pressure at which the column operates, the reflux ratio of the column, and whether the column has a full or partial condenser. The full range of key words is listed below. Again, only the stem of the words in the description are matched against this list.

- **partial** - sets a flag so that if **cond** is matched, then the column will use a partial condenser.
- **total** - as for **partial**, only with a total condenser.
- **cond** - if either the **partial** or **total** flags have been set, then the column condenser is set to be the relevant type. Otherwise nothing happens.
- **pres** - sets the pressure flag such that the next real number is set to be the pressure. The word after the real number is checked against a list of recognised units (see appendix F for list). If no recognisable units are matched then it is assumed that the pressure is given in bar.
- **heavy** - sets a flag such that the next component given is assumed to be the heavy key component. The flag also indicates that if **recov** is matched, then the next real number is set to be the heavy key recovery.
- **lvc** - identical to **heavy**.
- **light** - as for **heavy**, only for the light component.
- **mvc** - as for **light**.
- **reflux** - sets a flag such that the next real number is set to be the reflux.
- **recov** - sets a flag such that the next real number is set to be the recovery of either the heavy or light key components, dependent on what flags have been set.

Once the description has been parsed for the second time, the extracted data are entered into slots in the unit object. If any of the following slots do not exist then they are created by this method. If any slot exists and is filled, i.e. does not have status *unassigned*, then it is not overwritten. Otherwise the data are added to the slot with status *estimated*.

- **Name** - the name of the unit as given by the user. If no name is given, then one is generated. It should not be confused with the name of the unit object.
- **Function** - the unit function, a separator.
- **Condenser** - either total or partial. If no preference is stated then if the boiling point of the light key component, if given, is less than 250K, then a partial condenser is used. Otherwise a total condenser is specified.
- **Reflux_ratio** - the reflux ratio of the column. If no reflux ratio is given, then a reflux ratio of -1.3 is used. This follows the ASPEN convention of using a negative reflux ratio to denote a reflux ratio of that number multiplied by the minimum reflux ratio.
- **Heavy_key** - the heavy key component for the separation. If no heavy key has been explicitly given in the description, then the parser attempts to work it out. If two components have been given in the description, and the light key has been specified, then the other component is assumed to be the heavy key. If no keys but two components are specified in the description, then the boiling points of the two components are examined. The component with the lowest boiling point is assumed to be the light key, and the component with the highest boiling point is assumed to be the heavy key. If no components are given then this slot is left blank.
- **Heavy_key_recov** - the fraction of the heavy key component in the input stream which is recovered in the tops product. If no value is given, then a default value of 0.01 is used.
- **Light_key** - the light key component for the separation. If no light key has been specified then an attempt is made to estimate it in the same manner as for the heavy key.
- **Light_key_recov** - the fraction of the light key component in the input stream which is recovered in the tops product. If no recovery is specified, then a default of 0.99 is used.
- **Pressure** - the pressure at which the separation operates. If no pressure is given, then it is assumed that the separation takes place at atmospheric pressure.
- **Split_sharpness** - the sharpness of the split. For a distillation this is always set to sharp.

A.2 Splitter

A splitter unit takes one stream as an input and splits it into two or more output streams. Other than the division of flow, no other separation or processing is performed on the stream. To define a splitter, what is required are the fractions of the input stream

going to all but one of the output streams of the unit. If no output streams have been defined, then no fractions may be added.

No key words are actually matched here. Instead the sentence is parsed for: the unit name, product stream names (given in the product stream slot of the unit) and real numbers.

The product stream names and the real numbers are paired off in order of appearance to give the split fraction for that stream. Either the split fraction or the stream name may be given first. No checks are made to see if the split is feasible, i.e. to see if the fractions add up to more than one. This is left up to the user.

The following slots are either created or filled:

- **Name** - the unit name.
- **Function** - this will be a splitter.
- **StreamFrac_i** - this is the fraction of the input stream going to stream *i*. There may be as many of these slots as is desired, up to one less than the number of output streams, with the slot name changing to refer to the particular output stream.

A.3 Reactor

A reactor unit carries out one or more reactions. These reactions must be defined in reaction objects. For each reaction the reactor needs to know the name of the reaction object, and either an extent of reaction, or a conversion and a key component. Also the operating conditions of the unit, the temperature and pressure, are required. When parsing the description, it is assumed that only one reaction is being carried out in the reactor. If more than one reaction is present then the relevant slots and data for these reactions will have to be added manually.

The key words recognised here are:

- **extent** - sets a flag such that the next real number matched is the extent of reaction, in $kmolhr^{-1}$.
- **conversion** - sets a flag such that the next real number matched is the conversion.
- **temper** - sets a flag such that the next real number matched is the temperature at which the reactor operates. The word immediately after the number is examined for units (see appendix F for recognised units). If no units are given, then temperature is assumed to be given in Kelvin.

- **pres** - identical to **temper**, only for the reactor pressure. The default units for the pressure here is bar.

As well as the above data, the description is also parsed for reaction and component names. The reaction names must be the same as the object names, e.g. `100_22_react1_01.html`. However the component names may be written either as the object name cf. reaction objects, or by their name, e.g. `100_231_methane_1_01.html` or `methane`. It is assumed that any reaction object given in the description is the reaction taking place in the reactor. Any component object matched is assumed to be the key component for that reaction, unless an extent of reaction is given in the description in which case no key component is required.

The slots created or filled in the unit object by the parser for a reactor are:

- **Name** - the unit name.
- **Function** - this will be reactor here.
- **Reaction** - this slot contains the reaction object if matched in the description.
- **Conversion** - if a conversion was matched in the description, then it is entered here. If the word "conversion" was used in the description, but no actual conversion was matched then this slot will be created but left empty. Note that if an extent was specified or even mentioned in the description, then both this slot and the **Key_component** slot will not be created.
- **Key_component** - this slot contains the key component for the reaction if matched in the description.
- **Extent** - if a value for the extent was given in the description, then it is entered here. If the extent was only mentioned in the description, then this slot will be created empty. If the reaction conversion is given then this slot will not be created.
- **Temperature** - the temperature at which the reactor operates. If no temperature is given then it is assumed that the reactor operates at 300 K.
- **Pressure** - the pressure of the reactor. If no pressure is given then a default value of 1 bar is used.

If more than one reaction is present then the second reaction should be placed in a slot labelled `reaction_1`. A similar convention should be followed for the conversion, extent, etc.

A.4 Flash

A flash unit takes one input stream and separates it into two output streams by altering either the temperature or pressure of the input stream or both. Strictly speaking a flash should lower the pressure, but no checks are made on this by the parser.

Only two key words are recognised here:

- `temper` - sets a flag such that the next real number is the temperature at which the flash operates. The word immediately after the number is examined for units (see appendix F for units recognised). If no units are given, then it is assumed to be Kelvin.
- `pres` - identical to `temper`, only for the flash pressure. The default unit here is bar.

Also extracted from the description are any component names and the unit name. If either the temperature or the pressure are not given, then an attempt is made to estimate the unknown quantity. If two or more components are given to be separated, then the boiling points of the first two components given are obtained. If no boiling points can be obtained, then no calculations are performed. In this case if a pressure has been given, then the boiling points of the two components are estimated at the raised pressure using the following equation derived from a combination of the Clausius/Clapeyron equation and Trouton's Rule:

$$T = \frac{T_b}{1 - 0.1 \ln P} \quad (\text{A.1})$$

where:

T = temperature in Kelvin

T_b = atmospheric boiling point of component in Kelvin

P = saturated vapour pressure in bar

If the pressure given in the description is substituted for P , the saturated vapour pressure in the equation, together with the atmospheric boiling point of one of the given components, then the resultant calculation will provide an estimate for the raised pressure boiling point of that component.

The raised pressure boiling points are calculated for the two components and the arithmetic average of these temperatures is then used for the operating temperature of the flash unit.

If two components have been specified, and a temperature but no pressure, then a similar procedure is carried out to estimate the operating pressure of the flash unit.

The slots created or filled by the parser for a flash unit are:

- **Name** - the unit name.
- **Function** - this is automatically separator.
- **Temperature** - the temperature at which the unit operates as parsed or calculated. If no temperature is given or can be calculated then a default of 300 Kelvin is used.
- **Pressure** - the pressure at which the unit operates, as given or calculated. A default of 1 bar is used if no value is given or can be estimated.
- **Split_sharpness** - the split sharpness which for a flash is always non-sharp.

A.5 Mixer

A mixer unit takes two or more input streams and mixes them into a single output stream. No parameters are needed and thus no keywords are required. The only data that are extracted from the description is the unit name, if present.

The slots created or filled by the parser are:

- **Name** - the unit name.
- **Function** - this will be mixer for this unit.

A.6 Contactor, Stripper, Absorber

These three unit types are treated identically here when parsing their description. Each of these units has two input streams and two output streams. The parameters required here are the operating pressure, and the number of stages (if a stagewise column). The keywords recognised here are:

- **pres** - sets a flag such that the next real number matched is the operating pressure of the unit. The word immediately after the number is examined for units (see appendix F for units recognised). If no units are given then it is assumed that the pressure has been given in bar.

- **stage** - sets a flag such that the next real number matched is the number of stages for the unit.

The only other item of information extracted from the description is the unit name.

The following slots are created or filled by the parser:

- **Name** - the unit name.
- **Function** - the function of the column, either contactor, stripper or absorber, dependent upon what was initially matched.
- **Pressure** - the operating pressure of the unit. If no pressure is given, then it is assumed that the column operates at 1 bar.
- **Num_stages** - the number of stages of the unit.

A.7 Heater, Cooler

Heaters and coolers are parsed for the same key words. The units take a single input stream and adjust the temperature of the single output stream. The keywords recognised are:

- **temper** - sets a flag such that the next real number is the temperature at which the unit operates. The word immediately after the number is examined for units (see appendix F for units recognised). If no units are given, then it is assumed to be Kelvin.
- **pres** - identical to **temper**, only for the unit pressure. The default unit here is bar.

The only other item extracted from the description is the unit name.

The following slots are created and/or filled by the parser:

- **Name** - the unit name.
- **Function** - either heater or cooler, dependent on what was originally matched.
- **Pressure** - the operating pressure of the unit. If no pressure was specified, then a pressure of 1 bar is used.
- **Temperature** - the desired temperature of the output stream. If no temperature has been given in the description, then a value of 300 K is used here.

A.8 Vaporiser, Condenser

When parsing the description of a vaporiser or a cooler, the same key words are searched for. The units take a single input stream and produce either a single saturated vapour stream for a vaporiser, or a single saturated liquid stream for a condenser. The description is examined for only one keyword:

- **pres** - sets a flag such that the next real number is the pressure at which the unit operates. The word immediately after the number is examined for units (see appendix F for units recognised). If no units are given, then it is assumed to be bar.

Also extracted from the description is the unit name, if given.

The following slots are created and/or filled by the parser:

- **Name** - the name of the unit.
- **Function** - either vaporiser or condenser, dependent on what was originally matched.
- **Pressure** - the operating pressure of the unit. If no pressure was specified, then a pressure of 1 bar is used.

Appendix B

Generation of Equations for Unit Types

This appendix contains the linear equations generated for individual unit types. There are two types of equations: balance and specification. The component flows of one output stream for each unit are given by balance equations.

molar flow of component i out - total molar flow of component i in = moles i generated in unit

The only unit where components are generated or removed is a reactor, so the right hand side of this equation is normally zero. N.B. when writing the following equations, variables starting with x , e.g. x_1 , x_6 , etc. are actual equation variables. Any other variables are specification variables, and will be replaced by numbers in the actual equations.

If there is more than one output stream, then the component flows in these streams must be given by specification equations. Both equation types are discussed for the individual units below.

B.1 Feed

A feed stream is considered to be a unit with a single output stream for this analysis. The equations generated consist simply of the component molar flows of the feed stream.

$$\begin{aligned}x_1 &= f_1 \\&\vdots\end{aligned}$$

where:

x_1 = flow of component A in feed stream

f_1 = component flow

$\vdots$

Note that this equation will be repeated for components B, C, etc.

B.2 Mixer

As a mixer has only one output stream, all its equations are balance equations.

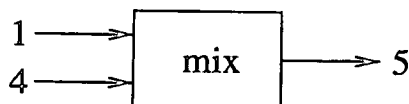


Figure B.1: A mixer unit

$$-x_1 - x_6 + x_{11} = 0$$

$\vdots$

where:

x_1 = flow of component A in stream 1

x_6 = flow of component A in stream 4

x_{11} = flow of component A in stream 5

B.3 Reactor

Like the mixer, a reactor also has only one output stream. However as material is being generated or removed here, then the right hand side of the balance equation is no longer zero.

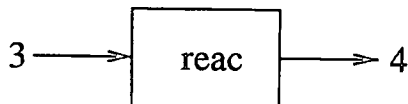


Figure B.2: A reactor unit

$$-x_3 + x_4 = \xi_A$$

$\vdots$

where:

x_3 = flow of component A in stream 3

x_4 = flow of component A in stream 4

ξ_A = amount of A created in reactor

When only one reaction takes place in the reactor, the right hand side of the equation is the extent of the reaction, multiplied by the reaction coefficient of the particular component to which the molar flows refer for that equation. If more than one reaction is taking place in the reactor then a value is worked out for each reaction, and these figures are added together. Where no extent has been given in the object for a reaction, it is evaluated it by obtaining the specified key component, its conversion, and the flow of the key component into the unit.

B.4 Splitter

The component flows of all but one of the output streams must be given by specification equations. The component flows of the remaining output stream are given by balance equations.

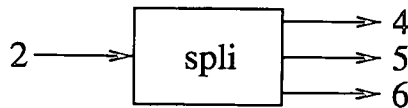


Figure B.3: A splitter unit

$$x_6 - \theta_1 x_1 = 0$$

$$x_{11} - \theta_2 x_1 = 0$$

$$-x_1 + x_6 + x_{11} + x_{16} = 0$$

$\vdots$

where:

x_1 = flow of component A in stream 2

x_6 = flow of component A in stream 4

x_{11} = flow of component A in stream 5

x_{16} = flow of component A in stream 6

θ_1 = fraction of stream 2 leaving in stream 4

θ_2 = fraction of stream 2 leaving in stream 5

The fractional recovery of the input streams to the various output streams may be given in the unit object in the manner as described in section 4.7.11. If no information is given in the unit object, then the split fractions are calculated using the stream flows to and from the unit.

B.5 Separator

The component flows for all but one of the output streams of the separator must be given by specification equations. The component flows for the remaining output stream will be given by the balance equation.

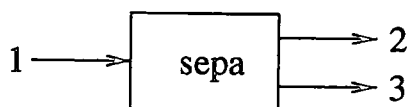


Figure B.4: A separator unit

$$\begin{aligned}
 x_6 - \theta_1 x_1 &= 0 \\
 -x_1 + x_6 + x_{11} &= 0 \\
 &\vdots
 \end{aligned}$$

where:

x_1 = flow of component A in stream 1

x_6 = flow of component A in stream 2

x_{11} = flow of component A in stream 3

θ_1 = fractional recovery of component A from stream 1 to stream 2

The fractional recovery of the various components are obtained from the input and output streams to the unit. If some of the recoveries cannot be calculated in this manner, then the components are arranged roughly in order of volatility, by sorting using their boiling point temperatures. Next it is seen if recoveries are increasing or decreasing with volatility. If decreasing then if a recovery of greater than 0.5 has been calculated for a component, then any unknown recoveries for more volatile components are assumed to be one, and unknown recoveries for less volatile components are assumed to be zero. If the recoveries are decreasing with volatility, then the opposite occurs.

B.6 Heater, Cooler, Vaporiser, Condenser

As the equations only perform a mass balance not an energy balance, and these units have a single input and a single output stream, then only balance equations are necessary.

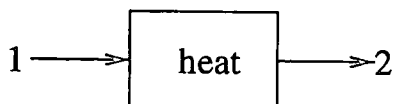


Figure B.5: A heater unit

$$-x_1 + x_6 = 0$$

$$\vdots$$

where:

x_1 = flow of component A in stream 1

x_6 = flow of component A in stream 2

B.7 Contactor, Stripper, Absorber

These three unit types are all modelled by the same equations. The flows of one of the output streams will be given by the balance equations, the component flows of the remaining streams by specification equations.

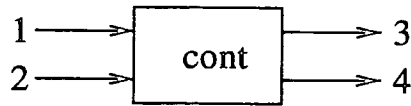


Figure B.6: A contactor unit

$$x_{11} - \theta_1 x_6 - \theta_1 x_1 = 0$$

$$-x_6 - x_1 + x_{11} + x_{16} = 0$$

$$\vdots$$

where:

x_1 = flow of component A in stream 1

x_6 = flow of component A in stream 2

x_{11} = flow of component A in stream 3

x_{16} = flow of component A in stream 4

θ_1 = fractional recovery of A in streams 1 and 2 to stream 3

The fractional recoveries are evaluated in a similar manner as for the separator unit. However for this unit the two input streams are first of all combined.

Appendix C

Class Files

Class files are simple ASCII files which are used to store the data for object classes: the methods, slots and the slot attributes. This appendix describes the structure of class files, and also contains the class files for the standard process engineering objects.

C.1 Class File Structure

The class data are stored internally in an ASCII file. This file consists of several sections, beginning with key words. These key words are listed below.

- `CLASS` - gives the name of the class.
- `INHERITED_FROM` - the name of the class from which this class inherits from.
- `LOGISTICAL_SLOTS` - the logistical slots.
- `OBJECT_SLOTS` - the class slots.
- `OBJECT_METHODS` - the class methods.
- `SELECTION_OPTIONS` - the selection options.
- `HISTORY_OPTIONS` - the history options.

Only the `CLASS` keyword is compulsory. The remaining keywords are optional. Each section of the class file should be separated by at least two new lines. The format of each of the individual sections is given below.

CLASS

On the line immediately following the CLASS keyword, the name of the class should appear. It must be composed of only alphanumeric characters, and the underscore ('_') character, and must be a single word. It is the only compulsory field in the class file, e.g.

CLASS

Component

INHERITED_FROM

This section indicates from what class the current class inherits from. The inheritance includes all slots (logistical and object) and the methods (including the selection options and history options). Any methods or slots of the same name in the class, overwrite those methods or slots inherited from the superclass. Only single inheritance is supported. On the line following the keyword, the name of the class from which this class inherits should be written, e.g.

INHERITED_FROM

Base_object

LOGISTICAL_SLOTS

Each of the logistical slots should be given in a separate line, starting on the line immediately after the keyword. Each line should contain:

- **Internal Name** - the name used to represent the slot internally. This will not be displayed on the HTML page.
- **Single/Multiple** - either single or multiple. This refers to whether the slot can simultaneously store more than one value.
- **Title** - the title of the slot as will be displayed on the HTML page. If the slot should not be displayed, then three dash marks (---) should be placed here.
- **Default Value** - if no value has been entered for the slot, then the default value will be printed on the HTML page.

The first three items on the line are compulsory, the default value is optional. If no default value is given, then an empty string is used when required, e.g.

```

filepath      single  ---
child  multiple      Children      'none created'

```

OBJECT_SLOTS

Each entry should be on a separate line and contain the following:

- **Name** - the slot name.
- **Size** - either single or multiple.
- **Slot Data** - a list of the slot attributes.

If one of the slot attributes is value, then description and status attributes are automatically added. To avoid this, a suffix may be added to the value attribute, consisting of either D and/or S to omit either the description or status or both, e.g. value[D]. However, if a value attribute is not present, and the description or status attributes are required, then a similar suffix may be added to any of the attributes to include these additional attributes. This is a short cut to writing the attributes out in the list. Examples of an entry are:

```

Component multiple component value[D] reac_co
Component multiple component[S] molec_wt

```

It is also possible to “nest” one slot inside another slot. This is accomplished by adding “//” to the end of the outer slot entry, followed by the nested slot definition. The nested slot definition is identical to that of an ordinary slot, e.g.

```

Phase multiple value // Component multiple component value[D] molec_wt mol_flow

```

The manner in which object slots are displayed depends upon the class of object to which they belong. The generic display method is for undefined value and description attributes to be displayed as text input boxes, and the remainder of the attributes to be displayed as fixed strings.

OBJECT_METHODS

In this section, an optional CGI script URL may be given on the same line as the OBJECT_METHODS keyword. This will be the script that will be used to process the

methods listed in this section. Alternatively, individual CGI scripts may be given in the method entry (see section 3.4.5 for the description of such a script). Each entry in the object methods section starts with a single dash (-), followed by a single word method name, and then a method description. The method description is that used on the HTML page for the method name. Optionally a CGI script may be given after this. Each new entry must start on a new line, although the method entry itself may be spread over several lines, e.g.

```
-update 'Update the object to register any manual changes'
```

SELECTION_OPTIONS

This section has the same format as the OBJECT_METHODS section.

HISTORY_OPTIONS

This section has the same format as the OBJECT_METHODS section.

Class files for the standard objects, and the base object from which they all inherit, are given in the following sections.

C.2 Base Object

CLASS

Base_object

LOGISTICAL_SLOTS

name_creator single ---

history_file single ---

filepath single ---

creator single Created_by

method single Last_applied_method 'new object'

parent single Created_from 'new object'

child multiple Children 'none created'

date single Created_when

address single Stored_where

OBJECT_METHODS

```
-update 'Update the object to register any manual changes'
```

```
-add_slot 'Add a slot to the object'
```

-remove_slot 'Remove a user defined slot from the object'

SELECTION_OPTIONS http://www.chemeng.ed.ac.uk/ross-bin/objects/gen_selections.pl

-mail 'Mail all objects to entered email address'

-new 'Create a new object (inc. different types)'

-delete 'Finish the session'

-upload 'Obtain an object from a URL'

-object 'Create a new custom class'

HISTORY_OPTIONS http://www.chemeng.ed.ac.uk/ross-bin/objects/gen_history.pl

-history 'View the history for this family'

-ancestor 'Show the history for all class objects'

-overall 'Show the history of all created objects'

-manifest_class 'Show logistical information for this class of object'

-manifest_overall 'Show logistical information for all classes of objects'

C.3 Component

CLASS

Component

INHERITED_FROM

Base_object

OBJECT_SLOTS

Name single value

Abs_formula single value

CARN single value

Aspen_name single value

Struct_formula single value

Molec_weight single value

Boiling_pt single value

Cp_a single value

Cp_b single value

Cp_c single value

Cp_d single value

Crit_temp single value

Crit_pres single value

Crit_vol single value

Gibbs_fe single value

Enthalpy_form single value

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/comp_methods.pl

-databank 'Get properties from databank'

-group 'Estimate properties using group contribution method'

-spread 'Create a spreadsheet containing the object data'

C.4 Mixture

CLASS
Mixture

INHERITED_FROM
Base_object

OBJECT_SLOTS
Component multiple component[S] molec_wt

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/mix_methods.pl
-add_comp 'Add component(s) to the object'
-remove_comp 'Remove component(s) from the object'
-spread 'Create a spreadsheet containing component data'
-properties 'View VLE data on selected components'

C.5 Stream

CLASS
Stream

INHERITED_FROM
Base_object

OBJECT_SLOTS
Phase multiple value // Component multiple component value[D] molec_wt mol_flow
mixture single value
Temperature single value
Pressure single value

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/stream_methods.pl
-add_comp 'Add component(s) to the object'
-remove_comp 'Remove component(s) from the object'
-spread 'Create a spreadsheet containing object data'
-addphase 'Add a new phase to the stream'

C.6 Reaction

CLASS

Reaction

INHERITED_FROM

Base_object

OBJECT_SLOTS

Component multiple component value[D] reac_co

mixture single value

Temperature single value

Gibbs_FE_R_298 single value

Enthalpy_R_298 single value

Gibbs_FE_R_T single value

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/react_methods.pl

-add_comp 'Add component(s) to the reaction'

-remove_comp 'Remove component(s) from the reaction'

-clear 'Clear other data slots'

-calculate 'Calculate reaction properties'

C.7 Unit

CLASS

Unit

INHERITED_FROM

Base_object

OBJECT_SLOTS

Feeds multiple stream stream_name

Products multiple stream stream_name

Description single value[D]

Name single value

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/unit_methods.pl

-aspen 'Create an ASPEN model of the unit'

-latest 'Use the latest version of referenced objects'

C.8 Topology

CLASS

Topology

INHERITED_FROM

Base_object

OBJECT_SLOTS

Name single value

Purpose single value

Feeds multiple stream stream_name

Products multiple stream stream_name

Units multiple unit unit_name

mixture single value[DS]

topology single value[DS]

top_infile single value[DS]

top_outfile single value[DS]

top_file single value[DS]

OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/top_methods.pl

-decompose 'Decompose single unit into two or more units'

-separate 'Separate units into different topology objects'

-combine 'Combine topology objects together'

-change 'Change stream/unit target'

-latest 'Use latest versions of objects'

-spread_model 'Create a spreadsheet model'

-return 'Return data from a solved spreadsheet model'

-aspen 'Create an ASPEN model'

-hysys 'Create a HYSYS model'

-summary 'Get data from an ASPEN model'

C.9 Equation

CLASS

Equation

INHERITED_FROM

Base_object

OBJECT_SLOTS

eqn_file single value[DS]

def_file single value[DS]

```
OBJECT_METHODS http://www.chemeng.ed.ac.uk/ross-bin/objects/equation\_methods.pl  
-solve 'Solve the equations'  
-edit 'Edit the equations'  
-delete_eq 'Delete some equations'  
-add_eq 'Add new equations/variables'
```

Appendix D

Object Data Storage

D.1 Introduction

Objects are stored as HTML files. This appendix will describe how the object data are contained within these files. There are also certain occasions where it is necessary to store object data in a file during method executions, e.g. when requesting extra information from the user. Rather than write the data in HTML format and then parse it again, it is written to an intermediate XML format which is easier to parse. It is more likely that a user will have to construct an XML file rather than an HTML file, e.g. in writing custom methods, thus a full description is given here of the XML format. A Document Type Definition (DTD) has been developed and is listed here with numerous comments. An example XML file has also been included.

D.2 HTML Files

Objects are stored as plain HTML files. Within these files, the object data are held within form elements. Two form elements in particular are used: the hidden element and the text box element. Hidden elements are used where the data are fixed and cannot be changed. Text box elements are used where the data have not yet been specified, or if it is specified, where it may be altered by the user, e.g. for slots with status *estimated*.

For each slot attribute, a single form element has to be used. The slot attributes for the data slots are given in the class files (see appendix C). Each of these slots is numbered from 0 upwards. The name of each form element holding slot attribute data consists of: the word “slot”, followed by the slot number, an underscore, and the attribute name. The name of the slot is considered to be a slot attribute, called “name”. For the value attribute, only the word “slot” and the slot number are required. For example, the

`Abs_formula` slot has attributes: name, value and description. This would be stored in an HTML object as:

```
<INPUT TYPE=HIDDEN NAME=slot1_name VALUE=Abs_formula>
<INPUT NAME=slot1 TYPE=HIDDEN VALUE="C7H8">
<INPUT NAME=slot1_description TYPE=HIDDEN VALUE="">
<INPUT NAME=slot1_status TYPE=HIDDEN VALUE="obtained from databank">
```

If the slot had been a multiple slot, e.g. mixture components, then the slot numbers for each of the multiple slots would have been 1a, 1b, etc. The final point to note is that if a slot contains another object, then this object is linked rather than included. This cuts down on the storage space required.

D.3 XML Files

When applying a method to an object, it is sometimes necessary to store the object data in a file, e.g. if extra information is required from a user. Rather than write the data in HTML format, and then parse it again, it is written to an intermediate XML format. A Document Type Definition (DTD) file has been written which is generic enough to display all of the objects. It is listed below with numerous comments.

- Root Element

```
<!ELEMENT class (class_name, data?, logistical?, methods?)>
```

Each XML file must have a root element. In this case it is called `class`. The list in the brackets shows what must be included in the XML file for it to be valid. These are all elements which are defined further on in the DTD. The question mark after some of the elements means that zero or one of this type of element must appear. There are a number of symbols used as quantifiers:

- * - the item must occur zero or more times
- + - the item must occur one or more times
- ? - the item occurs zero or one time
- (a | b) - either a or b occurs, but not both. This is usually followed by a *, meaning that any combination of any number of a or b is allowed.

The data, logistical and methods elements contain the same data that the corresponding sections contain in the HTML objects.

- Class Name

```
<!ELEMENT class_name (#PCDATA)>
```

The `class_name` element holds the name of the class. The `#PCDATA` means that the element must only contain ordinary text between the opening and closing tags, and no other elements, for the document to be valid.

- Data Tag

```
<!ELEMENT data (slot | nest)*>
<!ELEMENT nest (attribute+, slot*)>
<!ELEMENT slot (attribute)+>
<!ELEMENT attribute (#PCDATA)>
```

The data tag can contain a combination of two tags, either `slot` or `nest`. The `slot` tag contains the same data, or attributes as the HTML `slot`. The actual data for each attribute are contained between the opening and closing tags. The `nest` tag contains both `slot` and `attribute` tags, and is primarily designed to store the phase data for a stream.

One other point to note here is that for multiple slots, it is possible to use multiple tags with the same name, as long as the tags are not nested. Thus it is possible to have something like:

```
<slot name='component'>
<attribute name='name'>methane</attribute>
<attribute name='mol_flow'>45.7</attribute>
</slot>
<slot name='component'>
<attribute name='name'>hydrogen</attribute>
</slot>
```

Several attributes have been omitted from this example for clarity.

- Logistical Tag

```
<!ELEMENT logistical (log)*>
<!ELEMENT log (#PCDATA)>
```

The logistical tag contains just one tag, the `log` tag. This contains all of the logistical data for the object, e.g.

```
<log name='creator'>ross@chemeng.ed.ac.uk</log>
```

- Methods Tag

```
<!ELEMENT methods (object?, selection?, history*)>
<!ELEMENT object (item)*>
<!ELEMENT selection (item)*>
<!ELEMENT history (item)*>
<!ELEMENT item (#PCDATA)>
```

As in the HTML object, the methods are split into three separate sections. In this case, it means that the methods tag may contain an object tag, a selection tag, and a history tag. Each of these tags contains item tags, which detail the individual methods, etc.

◦ Attributes

Each of the tags may have certain attributes associated with them. These are defined in the DTD:

```
<!ATTLIST class_name name CDATA #REQUIRED>
<!ATTLIST attribute name CDATA #REQUIRED>
<!ATTLIST item name CDATA #REQUIRED
           url CDATA #IMPLIED>
<!ATTLIST log name CDATA #REQUIRED>
<!ATTLIST slot name CDATA #REQUIRED>
<!ATTLIST nest name CDATA #REQUIRED>
```

After the !ATTLIST declaration comes the tag name for the associated attribute. Next comes a series of:

- attribute name
- content, in this case Character DATA (CDATA)
- one of #REQUIRED, #IMPLIED, or #FIXED

Required means the attribute must be present, implied means that it is optional attribute and fixed means that its value is fixed.

The final section in the DTD contains a list of the standard entities. These simply refer to symbols which may not be used in certain situations in a valid XML document, e.g. a < sign. To accommodate this the ASCII number of the character may be used instead, prefixed with &# and followed by a semi-colon. Instead of using the ASCII numbers, other characters may be used. These are defined in the DTD as entities, e.g. for the greater than (>) symbol:

```
<!ENTITY gt      "&#62;">
```

One final point to note is that XML is case sensitive, and that all attributes must be enclosed in quotes. A complete example of an XML document is given in appendix D.5.

D.4 Document Type Definition

```
<!-- this file contains the first version of the DTD for the
      object classes
```

```

-->

<!-- data slot details -->
<!-- ELEMENT data (slot | nest)* -->
<!-- ELEMENT nest (attribute+, slot*) -->
<!-- ELEMENT slot (attribute)+ -->
<!-- ELEMENT attribute (#PCDATA) -->

<!-- logistical slot details -->
<!-- ELEMENT logistical (log)* -->
<!-- ELEMENT log (#PCDATA) -->

<!-- methods -->
<!-- ELEMENT methods (object?, selection?, history?) -->
<!-- ELEMENT object (item)* -->
<!-- ELEMENT selection (item)* -->
<!-- ELEMENT history (item)* -->
<!-- ELEMENT item (#PCDATA) -->

<!-- now for the attributes -->
<!-- ATTLIST class_name name CDATA #REQUIRED -->
<!-- ATTLIST attribute name CDATA #REQUIRED -->
<!-- ATTLIST item name CDATA #REQUIRED
      url CDATA #IMPLIED -->
<!-- ATTLIST log name CDATA #REQUIRED -->
<!-- ATTLIST slot name CDATA #REQUIRED -->
<!-- ATTLIST nest name CDATA #REQUIRED -->

<!-- and for the standard entities -->
<!-- ENTITY lt "&#38;#60;" -->
<!-- ENTITY gt "&#62;" -->
<!-- ENTITY amp "&#38;#38;" -->
<!-- ENTITY apos "&#39;" -->
<!-- ENTITY quot "&#34;" -->

```

D.5 Example Stream XML File

```

<?xml version="1.0"?>
<!DOCTYPE class SYSTEM
"http://www.chemeng.ed.ac.uk/people/ross/objects/class.dtd">

<class>

```

```
<class_name name="Stream"></class_name>
<data>
  <nest name="Phase">
    <attribute name="value">default</attribute>
    <attribute name="description"></attribute>
    <attribute name="status">estimated</attribute>
    <slot name="Component">
      <attribute name="component">99_302_acetone_1_03.html</attribute>
      <attribute name="value">99_302_acetone_1_03.html</attribute>
      <attribute name="status">unassigned</attribute>
      <attribute name="molec_wt"></attribute>
      <attribute name="mol_flow"></attribute>
    </slot>
    <slot name="Component">
      <attribute name="component">99_302_benzene_2_03.html</attribute>
      <attribute name="value">99_302_benzene_2_03.html</attribute>
      <attribute name="status">unassigned</attribute>
      <attribute name="molec_wt"></attribute>
      <attribute name="mol_flow"></attribute>
    </slot>
    <slot name="Component">
      <attribute name="component">99_302_chloroform_1_03.html</attribute>
      <attribute name="value">99_302_chloroform_1_03.html</attribute>
      <attribute name="status">unassigned</attribute>
      <attribute name="molec_wt"></attribute>
      <attribute name="mol_flow"></attribute>
    </slot>
  </nest>
  <slot name="mixture">
    <attribute name="value">99_302_mix1_01.html</attribute>
    <attribute name="description"></attribute>
    <attribute name="status"></attribute>
  </slot>
  <slot name="Temperature">
    <attribute name="value"></attribute>
    <attribute name="description"></attribute>
    <attribute name="status">unassigned</attribute>
  </slot>
  <slot name="Pressure">
    <attribute name="value"></attribute>
    <attribute name="description"></attribute>
    <attribute name="status">unassigned</attribute>
  </slot>
</data>
<logistical>
  <log name="name_creator"></log>
  <log name="history_file"></log>
  <log name="creator">ross@test.demo</log>
```

```

<log name="method">new object</log>
<log name="parent">new object</log>
<log name="date">Tue Aug 3 14:12:22 BST 1999</log>
<log name="address">
/home/html/htdocs/tmp/www.epee.ross.test.demo/99_302_top1_stream11_01.html
</log>
</logistical>
<methods>
<object>
<item name="update">Update the object to register any manual changes</item>
<item name="add_slot">Add a slot to the object</item>
<item name="remove_slot">Remove a user defined slot from the object</item>
<item name="add_comp">Add component(s) to the object</item>
<item name="remove_comp">Remove component(s) from the object</item>
<item name="spread">Create a spreadsheet containing object data</item>
<item name="addphase">Add a new phase to the stream</item>
</object>
<selection>
<item name="mail">Mail all objects to entered email address</item>
<item name="new">Create a new object (inc. different types</item>
<item name="delete">Finish the session</item>
<item name="upload">Obtain an object from a URL</item>
<item name="object">Create a new custom class</item>
</selection>
<history>
<item name="history">View the history for this family</item>
<item name="ancestor">Show the history for all class objects</item>
<item name="overall">Show the history of all created objects</item>
<item name="manifest_class">
Show logistical information for this class of object</item>
<item name="manifest_overall">
Show logistical information for all classes of objects</item>
</history>
</methods>
</class>

```

Appendix E

Miscellaneous Files

E.1 VLE Data Generation Input File

The following file is used to generate VLE data for benzene and toluene at a pressure of 20 bar.

TGS

IN-UNITS MET

DATABANKS PURECOMP / AQUEOUS / SOLIDS / INORGANIC / &
NOASPENPCD

PROP-SOURCES PURECOMP / AQUEOUS / SOLIDS / INORGANIC

COMPONENTS

benzene BENZENE/

toluene TOLUENE

PROPERTIES UNIQUAC

PROP-SET VLE MOLEFRAC TEMP UNITS='C' SUBSTREAM=MIXED PHASE=V &
L

PROP-TABLE PT-1 FLASHCURVE

MOLE-FLOW benzene 50 / toluene 50

STATE PRES=20 VFRAC=0.99999999

VARY MOLE-FRAC COMP=benzene

RANGE LOWER=0 UPPER=1 NPOINT=40

TABULATE PROPERTIES=VLE

Appendix F

Recognised Units

This section contains a list of the units recognised by the system.

tonne - metric tonne

mmHg - millimetres of Mercury

torr - torr

lbm - pounds mass

lbf - pounds force

ton - imperial ton

atm - atmosphere

bar - bar

psi - pounds per square inch

min - minute

day - day

cal - calorie

BTU - British Thermal Unit

Pa - pascal

in - inch

ft - foot

cm - centimetre

yd - yard

dm - decimetre

hr - hour

yr - year

kg - kilogramme

C - degree Centigrade

K - degree Kelvin

F - degree Fahrenheit

R - degree Rankine

g - gram

N - Newton

s - second

J - Joule

W - Watt

m - metre

l - litre

Aside from the above units, the following prefixes are also recognised:

m - milli

k - kilo

M - mega

Appendix G

API data

This appendix contains data relevant to the operation and use of the Application Programmers Interface (API). There are three sections: the API Fortran structures, the subroutines, and the method codes needed to apply methods.

G.1 API Fortran Structures

Several Fortran structures have been written for use with the API to complement the standard objects, which represent process entities. This section will contain a description of these structures. All of the structures contain other smaller structures. These constituent structures will be described first.

G.1.1 Parameters

All arrays in the structures are of fixed size. The size of these arrays are set by parameters, which are listed below.

Parameter	Description
<code>www_hash</code>	A parameter used to hash the arrays for ease of access.
<code>www_double_hash</code>	Another hashing parameter.
<code>www_max_att</code>	The maximum number of attributes of a slot.
<code>www_max_mult</code>	The maximum number of slots in a multiple slot.
<code>www_max_slots</code>	The maximum number of single data slots in an object.
<code>www_max_log</code>	The maximum number of single logistical slots in an object.
<code>www_max_slot_mult</code>	The maximum number of multiple data slots in an object.
<code>www_max_log_mult</code>	The maximum number of multiple logistical slots in an object.
<code>www_max_nest</code>	The maximum number of nested slots.

G.1.2 slot

```

type slot
    !stores a slot
    character*100 :: name, attribute(www_max_att), value(www_max_att)
    integer :: slot_num
end type slot

```

A slot structure holds the information on a slot. This forms an important part of most structures. The slot data fields are:

name - This is the name of the slot.

attribute - An array which contains the names of the slot attributes.

value - This array contains the values of the attributes given in the attribute array, e.g. `value(3)` contains the value for the attribute `attribute(3)`.

slot_num - The number of the slot in the object.

G.1.3 multiple

```

type multiple
    !a multiple slot
    character*100 :: name
    type (slot) :: data(www_max_mult)
    integer :: slot_num
end type multiple

```

A multiple structure contains multiple slot information. The data fields are:

name - The name of the slot.

data - An array of type **slot**, which contains the individual slot information.

slot_num - The number of the slot in the object.

G.1.4 nest

```
type nest
  character*100 :: name
  type (slot) :: nest_name, nest_sing(www_max_nest)
  type (multiple) :: nest_mult(www_max_nest)
  integer :: slot_num
end type nest
```

A **nest** structure is a special type which should only be used in special circumstances as its use can make a subsequent program quite complicated. The structure of a nested slot consists of an outer slot, together with one or more inner slots. The data fields are:

name - The name of the slot.

nest_name - A slot which contains the outer slot data.

nest_sing - An array of single slots which contains the single inner slots.

nest_mult - An array of multiple slots which contains the multiple inner slots.

slot_num - The slot number.

G.1.5 Object Structures

Object structures have been written for several classes of objects:

- component
- mixture
- stream
- reaction
- unit
- topology

They have all got identical fields, aside from the `stream` structure which has an additional field. Therefore only one structure will be examined in detail.

`type component`

```
!first of all for the arrays
type (slot) :: data_single(www_max_slots), log_single(www_max_slots)
type (multiple) :: data_multiple(www_max_slot_mult), &
    log_multiple(www_max_log_mult)

!need the url and address of the structure
character*200 :: url, address

!now for the www_hash lists
integer :: data_index(www_hash), log_index(www_hash)

!need to check if the object has been initialised
logical :: initialised
```

`end type component`

The data fields for the component structure are:

`data_single` - An array containing the single data slots.

`log_single` - An array containing the single logistical slots.

`data_multiple` - This array holds the multiple data slots.

`log_multiple` - The multiple logistical slots are contained in this array.

`url` - The url of the equivalent HTML object, if any, is held in this field.

`address` - The full address of the equivalent HTML object, if any, is contained in this field.

`data_index` - An array used to hash the single data slots.

`log_index` - This array is used to hash the single logistical slots.

`initialised` - Before using an object, all data fields must be initialised to null strings. This variable records whether this has been performed or not.

The only other data field present in any of the objects is the phase, which is in the `stream` structure:

```
!special case for the phases
type (nest) :: phases(www_max_nest)
```

The data field here is:

phases - An array which contains the phase information, for the stream.

G.2 API Fortran Subroutines

This section lists the various Fortran subroutines of the API, together with their parameters. Some of these subroutines perform error checking. If any error messages are generated, they are returned as a string, called **error**, by these subroutines. If no errors have occurred, then an empty string is returned. It is up to the user as to how these error messages are trapped and dealt with.

In almost all of the subroutine descriptions, at least one Fortran structure is required. In the examples a component structure is normally used. Unless otherwise stated, any of the structures representing standard process entities (stream, mixture, etc.) may be used instead.

G.2.1 `www_setup_api`

This subroutine starts a session for the user. It performs many of the same tasks as when a session is started via a browser (see section 3.3.1). A temporary directory is created as a work space for the user on the WWW server, and is populated with several objects. The name of this directory is based on the email address of the user. A new session identifier is also generated. If this directory already exists, then only a new session identifier is created by this subroutine. The subroutine call is:

```
subroutine www_setup_api(creator,filepath,url_path,error)
character(*), intent(in) :: creator
character(*), intent(out) :: filepath, error, url_path
```

The parameters are:

filepath - The filepath of the directory on the WWW server, corresponding to the temporary workspace of the user.

creator - The email address of the user.

url_path - The URL of the directory of the user's workspace.

error - Error message string.

G.2.2 `www_apply_method`

Calling this subroutine enables the user to apply a method to an object on the WWW server. In order to do this, the user must have created a structure, which must exist as an HTML object on the WWW server. An indicated method is applied to this object, and a new HTML object created on the WWW server. The data from this new object are returned by the server and used to create a new structure of the same type as the one to which the method was applied. Several methods require one or more parameters. These are given in an optional array. Finally if the method cannot be applied, then the subroutine returns an error message.

```
subroutine www_apply_method(method,input,output,error(,param))
character(*), intent(in) :: method, param(:)
type (component), intent(inout) :: input
type (component), intent(out) :: output
character(*), intent(out) :: error
```

The parameters are:

method - The code of the method to apply to the object. The method codes for each of the standard classes are listed in appendix G.3.

input - Any of the Fortran structures, in the above example it is a component. This is the object that the method will be applied to, and the only requirement is that it must exist as an HTML file on the server.

output - A Fortran structure, which must be the same type as input, to which the data in the new object created by applying the given method will be written to.

error - Error message string.

param - An optional array, and one that contains any parameters required to apply the method.

G.2.3 `www_write_html`

This subroutine takes one of the Fortran structures, and writes the equivalent HTML object on the server. The name of the new object will follow the naming scheme

described in section 3.3.3. If the `url` field in the structure is filled, then the new object will be a child of the old object. Otherwise a new family will be started. The URL and the address of the new object will be put into the `url` and `address` fields of the structure.

```
subroutine www_write_html(path,object,error)
character(*), intent(in) :: path
type (component), intent(inout) :: object
character(*), intent(out) :: error
```

The parameters here are:

`object` - A Fortran structure containing the object data.

`path` - The filepath on the server where the HTML files are written.

`error` - Error message string.

G.2.4 `www_get_obj_from_html`

The `www_get_obj_from_html` subroutine takes data from an HTML object, and writes the data to a Fortran structure.

```
subroutine www_get_obj_from_url(url,local_dir,object)
character(*), intent(in) :: url, local_dir
type (component), intent(out) :: object
```

The parameters used here are:

`url` - The URL of the HTML object.

`object` - The Fortran structure to which the data will be written.

`local_dir` - A directory where scratch files may be written

G.2.5 `www_initialise`

Before a structure is used, it must be initialised. This subroutine initialises a given structure, adding the standard slots for the equivalent object class to the structure, and setting all its data fields to either zero or null strings.

```
subroutine www_intialise(object)
type (component), intent(inout) :: object
```

The single parameter is:

object - The name of the structure to initialise.

G.2.6 `www_set_slot`

As well as manipulating the slot arrays directly, the slot data may be changed by using this subroutine. The advantage of using this subroutine is that the slot name may be given here, rather than the array index of the slot. In order to maintain the list of slots in the object, it is important that only the two subroutines: `www_add_comp` and `www_remove_comp` are used to alter the slot list.

```
subroutine www_set_slot(object, slot_name, slot_param, slot_value, error)
type (component), intent(inout) :: object
character(*), intent(in) :: slot_name, slot_param, slot_value
character(*), intent(out) :: error
```

The parameters here are:

object - The structure in which the slot will be set.

slot_name - The name of the slot.

slot_param - The parameter of the slot, e.g. name, value, description, etc.

slot_value - The value of the slot parameter.

error - Error message string.

G.2.7 `www_write_xml`

This subroutine writes the data in an object to a given file in XML format.

```
subroutine www_write_comp_xml(object, data_file)
type(component), intent(in) :: object
character(*), intent(in) :: data_file
```

The parameters are:

object - The structure containing the data.

data_file - The data file to write to.

G.2.8 `www_add_slot`

To enable slots to be accessed using the `www_set_slot` subroutine, this subroutine must be used to add slots to the structure.

```
subroutine www_add_slot(object, slot_name, error)
character*(*), intent(in) :: slot_name
type (component), intent(inout) :: object
character*(*), intent(out) :: error
```

The parameters are:

object - The object to add the slot to.

slot_name - The name of the slot.

error - Error message string.

G.2.9 `www_remove_slot`

As with the `www_add_slot` subroutine, this subroutine must be used to remove a slot from a structure if the `www_set_slot` is to be used to alter slot data.

```
subroutine www_remove_slot(object, slot_name, error)
character*(*), intent(in) :: slot_name
type (component), intent(inout) :: object
character*100, intent(out) :: error
```

The parameters here are:

object - The object to remove the slot from.

slot_name - The name of the slot.

error - Error message string.

G.2.10 `www_add_comp`

This subroutine adds a component to a structure, and so is only relevant to stream, reaction and mixture structures.

```
subroutine www_add_comp(object,attribute,value,error)
type (mixture), intent(inout) :: object
character*(*), intent(in) :: attribute(:), value(:)
character*(*), intent(out) :: error
```

The parameters in this subroutine are:

object - The structure to add the component to.

attribute - An array containing attribute names of the component to be added. One of these attributes must be `name`.

value - This array corresponds to the `attribute` array, and contains the attribute values, e.g. `value(2)` will contain the value for the attribute given in `attribute(2)`. One of these attribute - value pairs must be the name of the component to be added.

error - An error message string.

G.2.11 `www_get_mult_slot_attribute`

This subroutine enables the accessing of an attribute in a multiple slot.

```
subroutine www_get_mult_slot_attribute(object, slot_name, check_attribute, &
    check_value, attribute, value, error)
type(component), intent(in) :: object
character*(*), intent(in) :: slot_name, check_attribute, check_value, attribute
character*(*), intent(out) :: value, error
```

The subroutine parameters are:

object - The structure whose multiple slots will be searched.

slot_name - The name of the multiple slot from which the slot attribute will be accessed.

Method Code	Reference	Method Name	Parameters
update	4.2.1	Update the object to register any manual changes	None
add_slot	4.2.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
databank	4.2.4	Get properties from the databank	None
group	4.2.5	Estimate properties using group contribution method	None
spread	4.2.6	Create a spreadsheet containing the object data	None

Table G.1: Method codes and parameters for component objects

`check_attribute` - Multiple slots are composed of several single slots. As each of these slots have the same attribute names, it is necessary for an attribute and a value to be passed to this subroutine. The first individual slot within the multiple slot which has an attribute - value pair matching the supplied ones has its attribute list checked for the attribute whose value is required.

`check_value` - The other half of the attribute - value pair, used to determine which slot within the multiple slot to examine.

`attribute` - The name of the attribute for which the value is required.

`value` - The value of the attribute.

`error` - An error message string.

G.3 Method Parameters

This section lists the method code and parameters needed to apply methods to each of the standard objects using the API. where parameters are required, they are given as an ordered list, and should be included in the parameter array in this order.

Method Code	Reference	Method Name	Parameters
update	4.2.1	Update the object to register any manual changes	None
add_slot	4.2.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
add_comp	4.3.1	Add component to the object	1. Component object
remove_comp	4.3.2	Remove components from the object	1. Component object
spread	4.3.3	Create a spreadsheet containing component data	None
properties	4.3.4	View VLE data on selected components	1. Component object 2. Component object 3. Pressure (bar)

Table G.2: Method codes and parameters for mixture objects

Method Code	Reference	Method Name	Parameters
update	4.2.1	Update the object to register any manual changes	None
add_slot	4.2.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
add_comp	4.4.1	Add a component to the object	1. Component object
remove_comp	4.4.2	Remove a component from the object	1. Component object
spread	4.4.3	Create a spreadsheet containing the object data	None
addphase	4.4.4	Add a new phase to the stream	None

Table G.3: Method codes and parameters for stream objects

Method Code	Reference	Method Name	Parameters
update	4.2.1	Update the object to register any manual changes	None
add_slot	4.2.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
add_comp	4.5.1	Add a component to the reaction	1. Component object
remove_comp	4.3.2	Remove a component from the reaction	1. Component object
calculate	4.5.2	Calculate reaction properties	None
clear	4.5.3	Clear other data slots	None

Table G.4: Method codes and parameters for reaction objects

Method Code	Reference	Method Name	Parameters
update	A	Update the object to register any manual changes	None
add_slot	4.6.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
aspen	4.6.4	Create an ASPEN model of the unit	None
latest	4.6.5	Use the latest version of referenced objects	If no parameters are given, then all hyperlinks are updated. Alternatively, if hyperlink names (e.g. feed stream names) are given then only these are updated

Table G.5: Method codes and parameters for unit objects

Method Code	Reference	Method Name	Parameters
update	4.7.1	Update the object to register any manual changes	None
add_slot	4.2.2	Add a slot to the object	1. Slot Name
remove_slot	4.2.3	Remove a user defined slot from the object	1. Slot Name
decompose	4.7.4	Decompose single unit into two or more units	None
separate	4.7.5	Separate units into different topology objects	None
combine	4.7.6	Combine topology objects together	1. Topology object
change	4.7.7	Change stream/unit target	1. Link to change 2. New target object
latest	4.7.8	Use latest versions of objects	If no parameters are given, then all hyperlinks are updated. If individual hyperlinks are given as parameters, then only these are updated.
spread_model	4.7.9	Create a spreadsheet model	None
return	4.7.10	Return data from a solved spreadsheet model	1. Spreadsheet URL
aspen	4.7.11	Create an ASPEN model	None
hysys	4.7.12	Create a HYSYS model	None
summary	4.7.13	Get data from an ASPEN model	1. URL of ASPEN file

Table G.6: Method codes and parameters for topology objects

Appendix H

Parameter Extraction for Simulation Models

There are two types of simulation model that may be created from a topology object using this system: a HYSYS model or an ASPEN model. In order to create the simulation models, data must be extracted from the various objects making up the topology object. This appendix lists what data are required to create both types of simulation models. Firstly the extraction of component names is discussed, followed by unit connectivity. Next the obtaining of the feed stream parameters is described. Finally the parameters extracted for each individual unit type are detailed. Where possible, the parameter extraction is described for ASPEN and HYSYS together.

H.1 Component Names

The component names are obtained from the component objects in the topology mixture object. These objects are compared with the component objects in the topology feed and product streams to ensure that no component is omitted.

The extraction of the component names from the component objects is significantly different for the model types. Therefore they will be discussed separately.

H.1.1 ASPEN

Component objects contain a slot, `Aspen_name`, which holds the ASPEN databank name for the component. The ASPEN names are extracted for each component. If this slot is empty then an error message is given. As well as the ASPEN name, an alias is required for the component. This is the name by which the component will be referred to in

the input file. The alias is generated by removing both the prefix and the suffix from the component object name, e.g. the alias for 99_143_methane_1_03.html would be methane. The first eight characters of the component alias must be unique. If this is not the case for an alias generated in this manner, then the alias is prefixed with a, b, c, etc. until a unique alias is obtained.

H.1.2 HYSYS

There is no databank of HYSYS component names. Thus estimates must be made of their names by using the value given in the component name slot. This is only an educated guess, and may not be correct.

H.2 Unit Connectivity

Unit connectivity is obtained from the unit objects. For each unit object, the unit name, e.g. B1, the input stream names and the output stream names are extracted and written to file. No checks are made at this stage to see if the unit has valid numbers of feed or product streams.

H.3 Feed Streams

For each feed stream the molar component flowrates are extracted from the relevant stream object. The only other required information are the stream temperature and pressure. The temperature is converted to Kelvin, and the pressure to atmospheres for the ASPEN model, and to Centigrade and kPa respectively for the HYSYS model.

H.4 Unit Parameters

The final task is to add the unit parameters. Each of the units recognised by the parsing (see section 4.7.1) are discussed below. For many of the units, several restrictions are made on the number of input and output streams and on various parameters being present. These are required in order to generate the models using these methods. For many of these restrictions it would be possible to construct the models by hand without all of the parameters and without many of the stream restrictions.

H.4.1 Mixer

This is modelled in ASPEN and HYSYS by the MIXER model. No parameters are required. One input stream, and two or more output streams are required, otherwise an error message is generated.

H.4.2 Splitter

The splitter is modelled using either the ASPEN FSPLIT model or the HYSYS TEE. It requires the molar split fractions of all but one of the output streams, i.e. the fraction of the input stream going to the indicated output stream. The model requires one input stream and two or more output streams. There are two methods for specifying the output stream fractions in the unit object. If there are only two output streams, then the name of the stream whose split fraction is to be specified may be given in slot *Stream*, and the split fraction in the slot *Fraction*. The split fraction for output stream *i* may also be given in the slot *Streamfrac_i*. The stream names used here must be identical to the names of the unit output streams. If the unit has more than two output streams, then the second method must be used to give the molar split fractions for all but one output stream.

H.4.3 Reactor

A reactor is modelled using the RSTOIC model in ASPEN, or by using a CONVERSION REACTOR in HYSYS. It requires one input stream and one output stream, otherwise an error message will be generated. The parameters required are:

- the operating conditions - temperature and pressure.
- the reactions occurring - at least one reaction object must be specified in the unit object.
- for each reaction:

ASPEN - either the extent of the reaction, or a conversion and key component.

HYSYS - a conversion and a key component.

If any of these are missing, then an error message will be returned. If only one reaction is specified in the unit object, then it should be in the slot *Reaction*. If there is more than one reaction, then they should be in the slots *Reaction_1*, *Reaction_2*, etc. If an extent of reaction is given it should be in the slot *Extent*. Otherwise the conversion and a key component, in slots *Conversion* and *Key_component* respectively, should be specified. The key component must be a component present in the reaction and may

be specified by either the component object name, or the component alias. If there is more than one reaction present, then the same convention as for reaction objects should be followed, i.e `Extent_1` for reaction 1, `Extent_2` for reaction 2, etc.

H.4.4 Separator

A separator may be modelled by one of three models, dependent upon what parameters have been given in the unit object. If the slot `Split_sharpness` is defined as sharp, then either a rigorous distillation or short-cut distillation model is used. If the slot `Split_sharpness` is defined as non-sharp, or not defined, then a flash model is used. All separators should have one input stream and two output streams.

In ASPEN, the FLASH2 model is used to simulate a flash and needs two parameters, the temperature and pressure of operation. These should be given in the slots `Temperature` and `Pressure` respectively. In HYSYS, a flash is modelled using a VALVE followed by a SEPARATOR. The same parameters are required.

Sharp separations are modelled using a distillation column. The decision over whether to use a rigorous distillation model or a shortcut distillation model depends upon the parameters given in the unit object. If enough parameters have been given to use the rigorous distillation model, then it is used. Otherwise the short-cut distillation model is used.

In ASPEN the rigorous distillation column is modelled by the RADFRAC model, and the short cut distillation is modelled by the DSTWU model. The parameters required by the RADFRAC model are grouped into three categories:

- Distillate flowrate (`Dist_flow`), Bottoms flowrate (`Bot_flow`), Distillate flowrate to feed flowrate ratio (`Dist_vs_feed`), Bottoms flowrate to feed flowrate ratio (`Bot_vs_feed`)
- Reflux ratio (`Act_reflux`), Boilup ratio (`Boil_reflux`)
- Reflux flow (`Reflux_flow`), Boilup flow (`Boil_flow`), Condenser duty (`Cond_duty`), Reboiler duty (`Reb_duty`)

Two parameters are needed from separate groups in order to write a RADFRAC model. Also the number of stages (`Act_stages`) and the feed stage (`Feed_locatn`) must be given. If the number of stages has been given, but no feed stage, then the feed stage is estimated to be half way up the column.

In HYSYS the rigorous distillation column is modelled using the DISTILLATION COLUMN model. This requires the number of stages (`Act_stages`), the reflux ratio (`Act_reflux`), the condenser duty (`Cond_duty`), the pressure (`Pressure`) and the feed stage (`Feed_locatn`).

In a similar manner to the ASPEN model, if the feed stage has not been given, it is estimated to be half way up the column.

If not enough parameters have been given for a rigorous distillation model, then a short cut distillation model is used instead. This requires the heavy and light key components (**Heavy_key**, **Light_key**), their recoveries (**Heavy_key_recov**, **Light_key_recov**), the reflux ratio (**Reflux_ratio**), the operating pressure (**Pressure**) and whether or not a total or partial condenser will be used, given in the slot **Condenser**. This is modelled in ASPEN using the DSTWU model, and in HYSYS using the SHORT CUT DISTILLATION model.

H.4.5 Heater

This is a single input, single output unit and takes as a parameter the temperature of the output stream. It is modelled in both ASPEN and HYSYS using the **HEATER** model. The ASPEN model requires an additional parameter of the output stream pressure. The parameters are given in slots **Temperature** and **Pressure**.

H.4.6 Cooler

The cooler is modelled exactly the same in ASPEN as a heater. In HYSYS it is modelled using a **COOLER** model. It takes the same parameters as the heater model.

H.4.7 Vaporiser

A vaporiser takes a single parameter, the pressure of the output stream (in slot **Pressure**). It is a single input, single output unit, and is modelled in both ASPEN and HYSYS using a **HEATER** model.

H.4.8 Condenser

The condenser is modelled in ASPEN using a **HEATER** model, and in HYSYS using a **COOLER** model. Like the vaporiser, it takes a single parameter, the pressure of the output stream (in slot **Pressure**).

H.4.9 Contactor, Stripper and Absorber

All of these units are modelled using the RADFRAC model in aspen, and the ABSORBER unit in HYSYS. A two input, two output unit this requires the number of stages (*Act.Stages*) and the operating pressure (*Pressure*).

Appendix I

Case Study Data

This appendix contains data relevant to the case studies described in chapter 6. The various case studies are:

- creating a topology object, and hence ASPEN model, from scratch
- creating a spreadsheet model from an ASPEN model
- manipulating an ASPEN model from a Fortran program
- providing a user interface for a Fortran program

I.1 HCN plant ASPEN model

The ASPEN model generated using the system for a hydrogen cyanide plant is listed below.

```
;This ASPEN input file has been generated automatically  
;using the Edinburgh process objects system
```

```
IN-UNITS MET  
;metric units set as a default
```

```
COMPONENTS  
  ammonia AMMONIA /  
  methane METHANE /  
  hydrogen HYDROGEN-CYANIDE /  
  ahydroge HYDROGEN
```

FLOWSHEET

BLOCK B2 IN=2 OUT=3
BLOCK B3 IN=3 OUT=4 5
BLOCK B4 IN=5 OUT=6 7
BLOCK B1 IN=6 1 OUT=2

SIM-OPTIONS ENERGY-BAL=YES

PROPERTIES NRTL

;default property set which uses NRTL

STREAM 1

SUBSTREAM MIXED TEMP=300 PRES=0.986923266716013
mole-flow ahydroge 0 /
hydrogen 0 /
methane 100 /
ammonia 100

BLOCK B2 RSTOIC

PARAM TEMP=300 PRES=9.86923266716013
STOIC 1 MIXED ahydroge 3/
hydrogen 1/
methane -1/
ammonia -1
CONV 1 MIXED ammonia 0.9

BLOCK B3 DSTWU

PARAM LIGHTKEY=methane RECOVL=0.99 HEAVYKEY=ammonia &
RECOVH=0.01 PTOP=1 PBOT=1 RDV=1 RR=-1.3

BLOCK B4 DSTWU

PARAM LIGHTKEY=ammonia RECOVL=0.99 HEAVYKEY=hydrogen &
RECOVH=0.01 PTOP=1 PBOT=1 RDV=1 RR=-1.3

BLOCK B1 MIXER

I.2 Cyclohexane model data

This section contains data for the cyclohexane plant. The data include the input file of the ASPEN model used to create the spreadsheet, the topology generated from this ASPEN model, the ASPEN simulation results, and the spreadsheet results. Also included for ease of reference is the diagram of the ASPEN model.

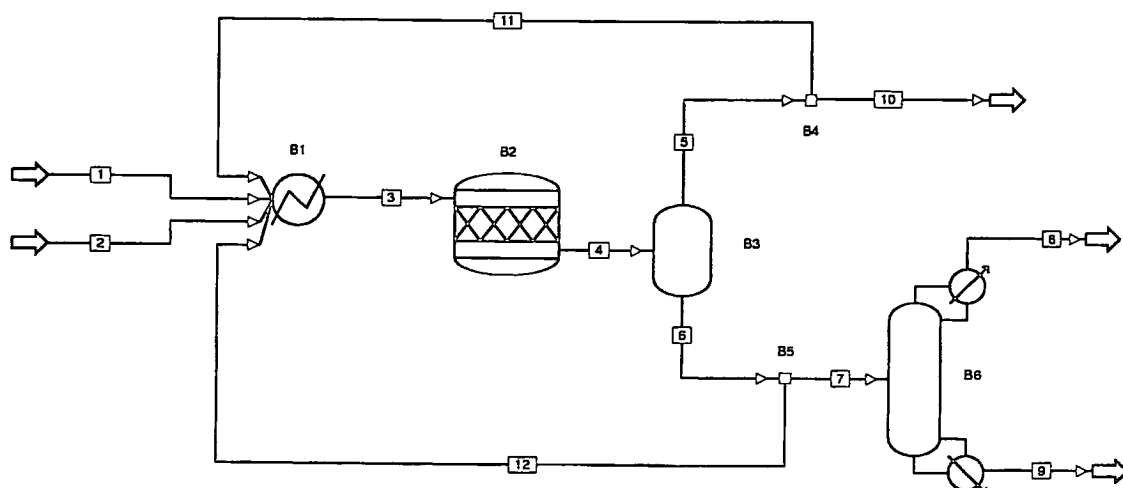


Figure I.1: Aspen model of a cyclohexane plant

I.2.1 ASPEN model input file

```
;
;Input Summary created by ASPEN PLUS Rel. 9.3-1 at 17:13:12 Mon May 28, 2001
;
```

```
TITLE 'HYDROGENATION OF BENZENE TO CYCLOHEXANE'
```

```
IN-UNITS MET VOLUME-FLOW='CUM/HR' ENTHALPY-FLO='MMKCAL/HR' &
HEAT-TRANS-C='KCAL/HR-SQM-K' PRESSURE=BAR TEMPERATURE=C &
VOLUME=CUM DELTA-T=C HEAD=METER MOLE-DENSITY='KMOL/CUM' &
MASS-DENSITY='KG/CUM' MOLE-ENTHALP='KCAL/MOL' &
MASS-ENTHALP='KCAL/KG' HEAT=MMKCAL MOLE-CONC='MOL/L' &
PDROP=BAR
```

```
DEF-STREAMS CONVEN ALL
```

```
DATABANKS PURECOMP / AQUEOUS / SOLIDS / INORGANIC / &
NOASPENPCD
```

```
PROP-SOURCES PURECOMP / AQUEOUS / SOLIDS / INORGANIC
```

```
COMPONENTS
```

```
HYDROGEN H2 HYDROGEN /
NITROGEN N2 NITROGEN /
METHANE CH4 METHANE /
BENZENE C6H6 BENZENE /
CYCLOHEX C6H12-1 CYCLOHEX
```

```
FLWSHEET
```

BLOCK B1 IN=1 2 11 12 OUT=3
BLOCK B2 IN=3 OUT=4
BLOCK B3 IN=4 OUT=5 6
BLOCK B4 IN=5 OUT=10 11
BLOCK B5 IN=6 OUT=7 12
BLOCK B6 IN=7 OUT=8 9

PROPERTIES RK-SOAVE

STREAM 1

SUBSTREAM MIXED TEMP=50 PRES=23
MOLE-FLOW HYDROGEN 136.5 / NITROGEN 0.7 / METHANE 2.8

STREAM 2

SUBSTREAM MIXED TEMP=35 PRES=1
MOLE-FLOW BENZENE 45

BLOCK B4 FSPLIT

FRAC 10 0.08

BLOCK B5 FSPLIT

FRAC 12 0.3

BLOCK B1 HEATER

PARAM TEMP=150 PRES=22.75

BLOCK B3 FLASH2

PARAM TEMP=50 PRES=21.4

BLOCK B6 RADFRAC

PARAM NSTAGE=15

FEEDS 7 8

PRODUCTS 8 1 V / 9 15 L

P-SPEC 1 13.8

COL-SPECS B:F=0.965 MOLE-RDV=1 MOLE-RR=1.2

BLOCK B2 RSTOIC

PARAM TEMP=200 PRES=21.7

STOIC 1 MIXED BENZENE -1 / HYDROGEN -3 / CYCLOHEX 1

CONV 1 MIXED BENZENE 0.998

I.2.2 Model Topology

feed

1

```
feed
2
heat    B1
1 2 11 12    3
reac    B2
3      4
sepa    B3
4      5 6
spli    B4
5      10 11
spli    B5
6      7 12
sepa    B6
7      8 9
end
product
10
product
8
product
9
```

I.2.3 ASPEN model solutions

Stream	1	2	3	4	5	6
From:			B1	B2	B3	B3
To:	B1	B1	B2	B3	B4	B5
Temperature C	50.0	35.0	150.0	200.0	50.0	50.0
Pressure BAR	23.0	1.0	22.75	21.7	21.4	21.4
Mole Flow KMOL/HR	140.0	45.0	247.223	112.409	45.968	66.441
Mass Flow KG/HR	339.697	3515.114	6051.037	6051.037	614.388	5436.649
Mole Flow KMOL/HR						
Hydrogen	136.5		154.046	19.232	18.994	0.239
Nitrogen	0.7		6.932	6.932	6.698	0.235
Methane	2.8		21.138	21.138	19.349	1.789
Benzene		45.0	45.028	0.09	0.001	0.089
Cyclohex			20.079	65.017	0.926	64.091

Stream	7	8	9	10	11	12
From:	B5	B6	B6	B4	B4	B5
To:	B6				B1	B1
Temperature C	50.0	-123.5	199.4	50.0	50.0	50.0
Pressure BAR	21.4	13.8	13.8	21.4	21.4	21.4
Mole Flow KMOL/HR	46.509	1.509	45.0	3.677	42.291	19.932
Mass Flow KG/HR	3805.654	23.829	3781.825	49.151	565.237	1630.995
Mole Flow KMOL/HR						
Hydrogen	0.167	0.167		1.52	17.474	0.072
Nitrogen	0.164	0.164		0.536	6.162	0.070
Methane	1.252	1.252		1.548	17.801	0.537
Benzene	0.062		0.062		0.001	0.027
Cyclohex	44.864	0.045	44.819	0.074	0.852	19.227

I.2.4 Spreadsheet Model

Data Area

Component	Molec_weight	Location	Name	Abs formula	CARN	Aspen name	Struct formula	Boiling_pt	Cp_a	Cp_b
Hydrogen	2	101_105_hydro	Hydrogen	H2	[001333-74-0]	HYDROGEN	H2	20.39999	27.14	0.008999988
nitrogen	28.01348	101_105_nitro	nitrogen	N2	[007727-37-9]	N2	0	77.344	0	0
Methane	16	101_105_meth	Methane	CH4	[000074-82-8]	METHANE	CH4	111.6	19.25	0.05199999
Benzene	78	101_105_benz	Benzene	C6H6	[000071-43-2]	BENZENE	C6H6	353.2	-50.77	0.5544
cyclohex	84.16128	101_105_cyclo	cyclohex	0	0	C6H12-1	0	353.87	0	0

Feeds and Operating Conditions

Please insert feed stream data here, together with the relevant unit parameters.

All relevant data will be copied to the stream table.

Component	1	2	Reaction	Recovery	5	6	Recovery
Hydrogen	136.5		-3	0.987596566			1
nitrogen	0.7			0.966170372			0.999923853
Methane	2.8			0.915387205			0.940678679
Benzene		45	-1	0.015008489			0
cyclohex			1	0.014246637			0
Total: kmol	140	45					
Total: te	0.337409436	3.51					
Feed	Feed	Of 3	From 4 to 5	Frac of 5 to 10:	Frac of 6 to 7:	From 7 to 8	
		Conversion of Benzene		0.08	0.7		
		0.998					

Stream Table

Component	1	2	11	12	3	Reaction	4	Recovery	5	6
Hydrogen	136.5	0	17.47549799	0.071569068	154.0470671	-3.00	19.23367007	0.9876	18.99510651	0.238563561
nitrogen	0.7	0	6.162095631	0.070356685	6.932452315	0.00	6.932452315	0.9662	6.697930033	0.234522282
Methane	2.8	0	17.80189217	0.536575447	21.13846761	0.00	21.13846761	0.9154	19.34988279	1.788584824
Benzene	0	45	0.001243472	0.026611233	45.0278547	-1.00	0.090055709	0.0150	0.0013516	0.088704109
cyclohex	0	0	0.852175311	19.22729863	20.07947394	1.00	65.01727293	0.0142	0.926277512	64.09099542
Total: kmol	140	45	42.29290457	19.93241106	247.2253156		112.4119186		45.97054844	66.4413702
Total: te	0.337409436	3.51	0.564220169	1.63096902	6.042598626		6.049846194		0.613282793	5.436563401
Feed	Feed	Bottoms of 5	Bottoms of 6	Ex-heat of 1	Of 3	Ex-react of 3	From 4 to 5	Tops of 4	Bottoms of 4	
					Conversion of Benzene					
					0.998					

Data Area							
Component	Cp_c	Cp_d	Crit temp	Crit pres	Crit vol	Gibbs fe	Enthalpy form
Hydrogen	-1.38E-05	7.59999E-09	33	13	65	0	0
nitrogen	0	0	0	0	0	0	0
Methane	0.0000119	-1.13E-08	190.4	46	99	-50.87	-74.9
Benzene	-1E-05	-0.000000095	562.2	48.9	259	121.68	80.77
cyclohex	0	0	0	0	0	0	0

Stream Table							
Component	10	5	7	6	Recovery	8	9
Hydrogen	1.519608521		0.166994493		1.0000	0.166994326	1.66994E-07
nitrogen	0.535834403		0.164165597		0.9999	0.164153097	1.25008E-05
Methane	1.547990623		1.252009377		0.9407	1.177738527	0.07427085
Benzene	0.000108128		0.062092876		0.0000	6.20929E-08	0.062092814
cyclohex	0.074102201		44.86369679		0.0000	4.48637E-05	44.86365193
Total: kmol	3.677643876		46.50895914			1.508930875	45.00002826
Total: te	0.049062623		3.805594381			0.023780085	3.781814296
	Tops of 5	Frac of 5 to 10:	Tops of 6	Frac of 6 to 7:	From 7 to 8	Tops of 7	Bottoms of 7
		0.0800		0.7000			

1.3 Distillation Column Data

The data used to manipulate an ASPEN program using the API is contained in this section. It includes the ASPEN model input file, the API Fortran program used to perform the manipulation, and the program results.

1.3.1 ASPEN Model Input File

```
;
;Input Summary created by ASPEN PLUS Rel. 9.3-1 at 14:55:59 Wed Apr 11, 2001
;
```

```

IN-UNITS MET VOLUME-FLOW='CUM/HR' ENTHALPY-FLO='MMKCAL/HR' &
      HEAT-TRANS-C='KCAL/HR-SQM-K' PRESSURE=BAR TEMPERATURE=C &
      VOLUME=CUM DELTA-T=C HEAD=METER MOLE-DENSITY='KMOL/CUM' &
      MASS-DENSITY='KG/CUM' MOLE-ENTHALP='KCAL/MOL' &
      MASS-ENTHALP='KCAL/KG' HEAT=MMKCAL MOLE-CONC='MOL/L' &
      PDROP=BAR

```

```

DEF-STREAMS CONVEN ALL

```

```

DATABANKS PURECOMP / AQUEOUS / SOLIDS / INORGANIC / &
      NOASPENPCD

```

```

PROP-SOURCES PURECOMP / AQUEOUS / SOLIDS / INORGANIC

```

```

COMPONENTS

```

```

      TOLUENE C7H8 TOLUENE /
      BENZENE C6H6 BENZENE

```

```

FLOWSHEET

```

```

      BLOCK B1 IN=1 OUT=2 3

```

```

PROPERTIES RK-SOAVE

```

```

STREAM 1

```

```

      SUBSTREAM MIXED TEMP=25 PRES=1
      MOLE-FLOW TOLUENE 50 / BENZENE 50

```

```

BLOCK B1 RADFRAC

```

```

      PARAM NSTAGE=30
      FEEDS 1 15
      PRODUCTS 2 1 L / 3 30 L
      P-SPEC 1 1
      COL-SPECS MOLE-RDV=0 MOLE-D=50 MOLE-RR=3

```

I.3.2 API program

```

1|program main
2|
3|use process_objects
4|implicit none
5|
6|!this script will upload an aspen backup file modelling a column
7|!separating an equimolar stream of toluene and benzene. The model
8|!will be simulated at three different pressure levels, and three

```

```

9|!different reflux rates
10|
11|character*200 :: error, creator, url_path, filepath
12|character*50 :: reflux, pressure
13|type (topology) :: top, top_out, top_end
14|integer :: i, j, out_ch
15|
16|!setup the API
17|creator = 'ross@chemeng.ed.ac.uk'
18|call www_setup_api(creator, filepath, url_path, error)
19|call check_error(error)
20|
21|!create the initial topology
22|call create_initial_topology(top, filepath, creator)
23|
24|!open up a file for the output
25|out_ch = 25
26|open(out_ch, FILE='out.dat')
27|write(out_ch,*) 'Pressure      Reflux      Tops Benzene Flowrate  &
28|      &Tops Toluene Flowrate'
29|write(out_ch,*) '   (bar)      Ratio      (kmol/hr)      &
30|      &      (kmol/hr)'
31|
32|do i = 5, 15, 5      !pressure
33|  do j = 3, 9, 3      !reflux
34|
35|    !need to convert integers into character strings
36|    write(pressure,'(I)') i
37|    write(reflux,'(I)') j
38|
39|    !add the pressure and reflux ratio to the model
40|    call add_details(top, top_out, reflux, pressure, url_path)
41|
42|    !create an aspen model and simulate it
43|    call simulate_model(top_out,top_end,'/home/ross/WWW/tmp/','&
44|      'http://www.chemeng.ed.ac.uk/people/ross/tmp/','&
45|      '/home/ross/tmp/API/')
46|
47|    !extract necessary data
48|    call extract_details(top_end,url_path,out_ch,pressure,reflux)
49|
50|  end do
51|end do
52|
53|!finally close the output file
54|close(out_ch)
55|
56|end program main

```

```
57|
58|!-----
59|
60|subroutine check_error(error)
61|  !this subroutine checks an error message and stops the program
62|  !execution if one is discovered
63|
64|  implicit none
65|  character*(*), intent(in) :: error
66|
67|  if(trim(adjustl(error)) /= '') then
68|    print *, 'error is: ', trim(error)
69|    stop
70|  end if
71|
72|end subroutine check_error
73|
74|!-----
75|
76|subroutine create_initial_topology(top, filepath, creator)
77|  !this subroutine will create the initial topology object,
78|  !including all of the stream, component and unit objects required
79|
80|  use process_objects
81|  implicit none
82|
83|  type (topology), intent(out) :: top
84|  character*(*), intent(in) :: filepath, creator
85|  type(topology) :: init_top
86|  character*200 :: param(5), error
87|
88|  !generate an empty topology
89|  call www_initialise(init_top)
90|
91|  !add in the filepath and creator slots
92|  call www_set_slot(init_top, 'creator', 'value', creator, error)
93|  call check_error(error)
94|  call www_set_slot(init_top, 'filepath', 'value', filepath, error)
95|  call check_error(error)
96|
97|  call www_write_html(filepath, init_top, error)
98|  call check_error(error)
99|
100|  !upload the aspen backup file
101|  param(1) = 'http://www.chemeng.ed.ac.uk/people/ross/benz_tol.bkp'
102|  call www_apply_method('summary', init_top, top, error, param)
103|  call check_error(error)
104|
```

```

105|end subroutine create_initial_topology
106|
107|!-----
108|
109|subroutine add_details(top, top_out, reflux, pressure, url_path)
110|  !this subroutine will add the details to the given topology model
111|
112|  use process_objects
113|  implicit none
114|
115|  type (topology), intent(inout) :: top
116|  type (topology), intent(out) :: top_out
117|  character*(*), intent(in) :: reflux, pressure, url_path
118|
119|  character*200 :: unit_url, error
120|  type (unit) :: block, block_obj
121|
122|  !alter data on the unit structure
123|  !need to create a unit structure here first from the object
124|  !get the URL of the unit
125|  call www_get_mult_slot_attribute(top, 'Units', 'unit_name', &
126|    'B1', 'unit', unit_url, error)
127|  call check_error(error)
128|  unit_url = trim(url_path) // trim(adjustl(unit_url))
129|
130|  !create the unit structure
131|  call www_get_obj_from_html(unit_url, './', block)
132|
133|  !set the pressure
134|  call www_set_slot(block, 'Pressure', 'value', pressure, error)
135|  call check_error(error)
136|
137|  !set the reflux ratio
138|  call www_set_slot(block, 'Reflux_ratio', 'value', reflux, error)
139|  call check_error(error)
140|
141|  !update the html unit object
142|  call www_apply_method('update',block, block_obj, error)
143|  call check_error(error)
144|
145|  !update the topology to link to the new unit object
146|  call www_apply_method('latest',top,top_out,error)
147|  call check_error(error)
148|
149|end subroutine add_details
150|
151|!-----
152|

```

```

153|subroutine simulate_model(top, top_out, web_dir, web_url, temp_dir)
154| !this routine will create and run an aspen model for the topology
155|
156| use process_objects
157| implicit none
158|
159| type (topology), intent(inout) :: top
160| type (topology), intent(out) :: top_out
161| character*(*), intent(in) :: web_dir, web_url, temp_dir
162|
163| character*200 :: error, url, command*500, param(5)
164|
165| !create the aspen model
166| call www_apply_method('aspen',top,top_out,error)
167| call check_error(error)
168|
169| !get the model input file name generated by the above method
170| url = top_out % url
171| url = url(1:scan(url,'.',TRUE.)) // 'inp'
172|
173| !copy it to the local temporary directory
174| command = 'perl -MLWP::Simple -e "getstore(' // "" // &
175|          trim(url) // ",'" // trim(temp_dir) // "tmp.inp'" // ""
176| call system(command)
177|
178| !set up aspen and simulate the results
179| command = "bash -c 'setup aspen; cd " // trim(temp_dir) &
180|          // "; aspen tmp.inp tmp'"
181| call system(command)
182|
183| !copy the input and summary so they may be accessed via the web
184| call system('cp ' // trim(temp_dir)// "tmp.inp " //trim(web_dir))
185| call system('cp ' // trim(temp_dir)// "tmp.sum " //trim(web_dir))
186|
187| !delete the remaining files
188| call system('rm -f ' // trim(temp_dir) // 'tmp.*')
189|
190| !extract the data from the aspen model
191| param(1) = trim(web_url) // 'tmp.sum'
192| call www_apply_method('summary', top, top_out, error, param)
193| call check_error(error)
194|
195|end subroutine simulate_model
196|
197|!-----
198|
199|subroutine extract_details(top, url_path, out_ch, pressure, reflux)
200| !this routine will extract the details from the aspen model

```

```

201|
202|  use process_objects
203|  implicit none
204|
205|  type (topology), intent(in) :: top
206|  character*(*), intent(in) :: url_path, pressure, reflux
207|  integer, intent(in) :: out_ch
208|
209|  character*200 :: stream_url, error, tol_flow, benz_flow
210|  type (stream) :: stream_obj
211|
212|  !want to extract the tops stream flowrates
213|
214|  !get the URL of the tops stream
215|  call www_get_mult_slot_attribute(top, 'Products', 'stream_name', &
216|    '2', 'stream', stream_url, error)
217|  call check_error(error)
218|  stream_url = trim(url_path) // trim(adjustl(stream_url))
219|
220|  !create a stream object
221|  call www_get_obj_from_html(stream_url, './', stream_obj)
222|
223|  !extract the flowrates
224|  tol_flow = stream_obj .COMP. 'toluene' .ATTRIBUTE. 'mol_flow'
225|  benz_flow = stream_obj .COMP. 'benzene' .ATTRIBUTE. 'mol_flow'
226|
227|  !pretty output
228|  write(out_ch, '(2(A,A2),2(A,A7))') ' ', &
229|    trim(adjustl(pressure)), ' ', trim(adjustl(reflux)), &
230|    ' ', trim(benz_flow), ' ', &
231|    trim(tol_flow)
232|
233|end subroutine extract_details
234|
235|!-----

```

I.3.3 Output of API Program

Pressure (bar)	Reflux Ratio	Tops Benzene Flowrate (kmol/hr)	Tops Toluene Flowrate (kmol/hr)
5	3	49.9518	0.04813
5	6	49.9894	0.01057
5	9	49.9940	0.00593
10	3	49.8813	0.11861
10	6	49.9722	0.02770

10	9	49.9839	0.01601
15	3	49.8181	0.18184
15	6	49.9549	0.04503
15	9	49.9733	0.02664

I.4 HCN Reactor Data

This section contains data used to create an interface to a hydrogen cyanide reactor program. It includes the Reactor class file, and the CGI script used to execute the Fortran program.

I.4.1 Reactor Class File

```
CLASS
Reactor
```

```
INHERITED_FROM
Unit
```

```
OBJECT_SLOTS
Pressure single value
Temperature single value
Tube_diam single value
Tube_length single value
Num_tubes single value
```

```
OBJECT_METHODS
-model 'Model an HCN reactor'http://www.chemeng.ed.ac.uk/ross-bin/demonstrate.pl
```

I.4.2 Reactor Custom Method CGI Script

```
1|#!/usr/local/bin/perl
2|
3|#this script will run a fortran program which models an HCN reactor
4|
5|#include the relevant perl modules
6|use CGI;                                #decode CGI data
7|use LWP::Simple;                         #download from URLs
8|use lib '/home/ross/objects/modules';
9|use Conversion;                          #convert between units
```

```

10|
11|#process the form and extract the CGI parameters
12|my $query = CGI->new();
13|my $xml_url = $query->param('xml_file');
14|my $url_path = $query->param('url_path');
15|
16|#need a temporary directory to store files, accessible by WWW
17|my $temp_dir = '/home/ross/WWW/tmp/';
18|
19|#download the xml file of the reactor, using 'get' routine from LWP::Simple
20|my $reactor = get($xml_url)
21|    or &return_error("Unable to download URL: $xml_url");
22|
23|#extract the 'Feeds' attributes from the reactor using the subroutine
24|#get_slot_attributes. This returns the attributes in a hash, basically
25|#an array accessed by strings rather than numbers
26|my %stream = &get_slot_attributes($reactor,'Feeds',
27|
28|    "A feed stream must be present");
29|#if a feed stream has been found, then download it
30|my $stream = ();
31|if(defined $stream{'stream'}) { #download the xml stream
32|    $stream{'stream'} =~ s/html$/xml/;
33|    $stream{'stream'} = $url_path.$stream{'stream'};
34|    $stream = get($stream{'stream'})
35|        or &return_error("Unable to download URL: $stream{'stream'}");
36|}
37|else {
38|    &return_error("No feed stream defined in reactor");
39|}
40|
41|#get a temporary file to store the input data in for the fortran program
42|#want this file to have a more or less unique name
43|$xml_url =~ m!.*!./(.*)!;
44|my $reactor_name = $1;
45|my $temp_file = $temp_dir.$reactor_name.time().rand().'.dat';
46|open(TMP,">$temp_file") or die "error in opening $temp_file: $!";
47|
48|#will need the reactor pressure
49|my %pressure = &get_slot_attributes($reactor,'Pressure',
50|    "Pressure must be given");
51|
52|#get the pressure units. If none given then assume it is bar
53|my $pres_units = ($pressure{'description'} or 'bar');
54|
55|#convert pressure to bar from whatever units given.
56|Conversion->convert($pressure{'value'},$pres_units,'bar');
57|

```

```
58|#add the stream parameters to the input file....
59|&add_stream_data(*TMP,$stream,$pressure{'value'});
60|
61|#....and the reactor parameters
62|&add_reactor_data(*TMP,$reactor);
63|
64|#close the input file and run the fortran program
65|close(TMP);
66|my $results = '/usr/local/bin/ssh loyne
67|    /home/ross/objects/demonstrations/reactor_model/hcn_react $temp_file';
68|
69|#check to see if fortran execution actually worked
70|if($?) {
71|    &return_error("Error in running fortran program: $?");
72|}
73|
74|#the results contains the mole flows of the components
75|#put these results into a hash for ease of access
76|$results =~ s/^\s+//; $results =~ s/\s+$//;
77|my @mole_flow = split(/\s+/, $results);
78|my %mole_flow = ();
79|$mole_flow{'ammonia'} = $mole_flow[0];
80|$mole_flow{'methane'} = $mole_flow[1];
81|$mole_flow{'hydrogen_cyanide'} = $mole_flow[2];
82|$mole_flow{'hydrogen'} = $mole_flow[3];
83|$mole_flow{'nitrogen'} = $mole_flow[4];
84|
85|#add the mole flows to the reactor output stream file
86|#this file should have a more or less unique name
87|my $prod_name = $reactor_name.time().rand().'.stream.xml';
88|my $prod_file = $temp_dir.$prod_name;
89|my $output = &add_mole_flow(\%mole_flow,$stream, $prod_file, $temp_dir);
90|
91|#create the reactor xml file - again with unique name
92|my $reactor_xml = $temp_dir.$reactor_name.time().rand().'.reactor.xml';
93|&create_xml_reactor_file($reactor,$prod_name, $reactor_xml);
94|
95|#change output xml file name to URL
96|my $url = $reactor_xml;
97|$url =~ s#/home#http://www.chemeng.ed.ac.uk/people#;
98|$url =~ s#WWW/#;
99|
100|#return URL
101|print "content-type: test/plain\n\n$url";
102|
103|#####
104|
105|sub add_stream_data
```

```

106|{
107|    #this routine will add the stream data to the data file
108|    my ($filehandle, $stream, $pressure) = @_;
109|
110|    #need to get the molar flows of each of the components
111|    my %flow = &get_flow($stream);
112|
113|    #add up all of the individual flows to get the total flow
114|    my $total = 0.0;
115|    foreach my $key (keys %flow)
116|    {
117|        $total += $flow{$key};
118|    }
119|
120|    #make sure that a flow has been obtained
121|    if($total < 0.0001) {
122|        &return_error("Flow is too small");
123|    }
124|
125|    #give the partial pressures and the total mole flow to the data file
126|    my $mult = $pressure/$total;
127|    print $filehandle $flow{'ammonia'}*$mult,"\n";
128|    print $filehandle $flow{'methane'}*$mult,"\n";
129|    print $filehandle $flow{'hydrogen_cyanide'}*$mult,"\n";
130|    print $filehandle $flow{'hydrogen'}*$mult,"\n";
131|    print $filehandle $flow{'nitrogen'}*$mult,"\n";
132|    print $filehandle $total,"\n";
133|}
134|
135|#####
136|
137|sub add_reactor_data
138|{
139|    #this routine will add the reactor data to the data file
140|    my($filehandle,$reactor) = @_;
141|
142|    #tube diameter in metres
143|
144|    #get tube attributes
145|    my %tube_diam = &get_slot_attributes($reactor,'Tube_diam',
146|                                         "Tube diameter must be given");
147|
148|    #get units
149|    my $diam_units = ($tube_diam{'description'} or 'm');
150|
151|    #convert tube diameter to be in metres
152|    Conversion->convert($tube_diam{'value'},$diam_units,'m');
153|

```

```

154|    #add to data file
155|    print $filehandle "$tube_diam{'value'}\n";
156|
157|    #similarly for tube length in metres.....
158|    my %tube_length = &get_slot_attributes($reactor,'Tube_length',
159|                                           "Tube length must be given");
160|    my $length_units = ($tube_length{'description'} or 'm');
161|    Conversion->convert($tube_length{'value'},$length_units,'m');
162|    print $filehandle "$tube_length{'value'}\n";
163|
164|    #....and for number of tubes.....
165|    my %tube_num = &get_slot_attributes($reactor,'Num_tubes',
166|                                         "Number of tubes is required");
167|    print $filehandle "$tube_num{'value'}\n";
168|
169|    #.....and for temperature in Kelvin
170|    my %temperature = &get_slot_attributes($reactor,'Temperature',
171|                                           "Temperature must be given");
172|    my $temp_units = ($temperature{'description'} or 'K');
173|    Conversion->convert($temperature{'value'},$temp_units,'K');
174|    print $filehandle "$temperature{'value'}\n";
175|}
176|
177|#####
178|
179|sub add_mole_flow
180|{
181|    #this routine will add the molar flows from the reactor to the new
182|    #output stream
183|    my ($flow_ref,$stream,$prod_file,$temp_dir) = @_;
184|
185|    #extract stream data
186|    my ($creator, $filepath) = ();
187|
188|    #get the names of the component objects, and the creator and
189|    #filepath from the stream data
190|    my %comp = &extract_stream_data($stream, \$creator, \$filepath);
191|
192|    #make sure all components are created objects
193|    foreach my $key (keys %$flow_ref)
194|    {
195|        unless (defined $comp{$key}) {
196|            #there is no component object for this component
197|            #create an XML file containing the component data
198|
199|            #refer to the component with an xml extension
200|            $comp{$key} = $key.'.html';
201|

```

```

202|         #get the XML file name
203|         my $comp_file = $temp_dir.$key.'.xml';
204|
205|         #write the general XML introduction
206|         my $filehandle = &write_xml_intro($comp_file,'component');
207|
208|         #add the known component data
209|         print $filehandle '<slot name="Name">','\n';
210|         print $filehandle '<attribute name="value">','ucfirst $key
211|             , "</attribute>\n";
212|         print $filehandle '<attribute name="description">'
213|             , "</attribute>\n";
214|         print $filehandle '<attribute name="status">entered by user'
215|             , "</attribute>\n</slot>\n</data>\n<logistical>\n";
216|         print $filehandle '<log name="filepath">',$filepath,"</log>\n";
217|         print $filehandle '<log name="creator">',$creator,"</log>\n";
218|         print $filehandle "</logistical>\n</class>\n";
219|         close($filehandle);
220|     }
221| }
222|
223| #write the XML introduction for the output stream XML file
224| my $filehandle = &write_xml_intro($prod_file,'stream');
225|
226| #add standard phase data
227| print $filehandle <<"NEXT";
228| <nest name="Phase">
229| <attribute name="value">default</attribute>
230| <attribute name="description"></attribute>
231| <attribute name="status">estimated</attribute>
232| NEXT
233|
234| #add the component data
235| foreach my $comp (keys %$flow_ref)
236| {
237|     print $filehandle "<slot name=\"Component\">\n";
238|     print $filehandle '<attribute name="component">',$comp{$comp}
239|         , "</attribute>\n";
240|     print $filehandle '<attribute name="value">',$comp{$comp}
241|         , "</attribute>\n";
242|     print $filehandle "<attribute name=\"status\">estimated"
243|         , "</attribute>\n";
244|     print $filehandle '<attribute name="mol_flow">',$flow_ref{$comp}
245|         , "</attribute>\n";
246|     print $filehandle "</slot>\n";
247| }
248|
249| #finish the phase, and add the known logistical data

```

```

250|    print $filehandle <<"END";
251|</nest>
252|</data>
253|<logistical>
254|<log name="filepath">$filepath</log>
255|<log name="creator">$creator</log>
256|</logistical>
257|</class>
258|END
259|    close($filehandle);
260|}
261|
262|#####
263|
264|sub create_xml_reactor_file
265|{
266|    #this routine will create the output reactor xml file, by removing
267|    #some logistical data from the input xml file, and adding the
268|    #new product stream name
269|    my ($reactor,$prod_name, $reactor_xml) = @_;
270|
271|    #remove any product streams from the reactor, and any children
272|    $reactor =~ s!<slot\s+name\s*=\s*\s*"Products\s">.*?</slot>!!igs;
273|    $reactor =~ s!<log\s+name\s*=\s*\s*"child\s">.*?</log>>!!igs;
274|
275|    #add in new product stream
276|    $prod_name =~ s/xml$/html/;
277|    $reactor =~ s!</slot>!
278|        </slot>
279|        <slot name="Products">
280|        <attribute name="stream">$prod_name</attribute>
281|        <attribute name="stream_name">product</attribute>
282|        </slot>!isx;
283|
284|    #finally write the reactor xml file and return the location of the xml fi
285|    open(TMP,">$reactor_xml") or die "error in opening $reactor_xml: $!";
286|    print TMP $reactor;
287|    close(TMP);
288|}
289|
290|#####
291|
292|sub return_error
293|{
294|    #this routine will process any error messages
295|    #if any HTML code is returned, then it is assumed that this is
296|    #an error message and displayed to the user
297|    my ($message) = @_;
```

```

298|
299|     print <<"END";
300|content-type: text/html
301|
302|<HTML>
303|<HEAD>
304|<TITLE>An Error Has Occurred</TITLE>
305|</HEAD>
306|
307|<BODY BGCOLOR=white>
308|<H1>An Error Has Occurred</H1>
309|<P>$message
310|</BODY>
311|</HTML>
312|END
313|     die();
314|}
315|
316|#####
317|
318|sub get_slot_attributes
319|{
320|    #this routine will return the attributes from a slot in a hash
321|    my ($xml, $slot, $error) = @_;
322|
323|    my %attribute = ();
324|
325|    if($xml =~ m!<slot\s+name\s*=\s*"${slot}">(.*?)</slot>!is) {
326|        my $data = $1;
327|        while($data =~ m!<attribute\s+name\s*=\s*"(.*)">(.*?)</attribute>!isgx) {
328|            $attribute{$1} = $2;
329|        }
330|    }
331|
332|    else {
333|        &return_error($error);
334|    }
335|
336|    return %attribute;
337|}
338|
339|#####
340|
341|sub extract_stream_data
342|{
343|    #this routine will extract data from the xml stream file
344|    my ($stream, $creator, $filepath) = @_;
345|

```

```

346|    my %comp = ();
347|
348|    #extract the component names and the component object names
349|    while($stream =~ m!<slot\s+name\s*=\s*"Component">(.*?)</slot>!igs) {
350|        my $data = $1;
351|
352|        $data =~ m!<attribute\s+name\s*=\s*"component">
353|            (\d+_ \d+_ (.*?) _ \d+_ \d+ \.html)</attribute>!isx;
354|        $comp{$2} = $1;
355|    }
356|
357|    #get the creator and filepath attributes also
358|    if($stream =~ m!<log\s+name\s*=\s*"creator">(.*?)</log>!igs) {
359|        $$creator = $1;
360|    }
361|    if($stream =~ m!<log\s+name\s*=\s*"filepath">(.*?)</log>!igs) {
362|        $$filepath = $1;
363|    }
364|
365|    return %comp;
366|}
367|
368|#####
369|
370|sub write_xml_intro
371|{
372|    #this routine will write the introduction to an xml file, and
373|    #return the filehandle
374|    my ($file, $class) = @_;
375|
376|    open(FILE,">$file") or die "error in opening $file: $!";
377|
378|    print FILE <<"START";
379|<?xml version="1.0"?>
380|<!DOCTYPE class SYSTEM
381|"http://www.chemeng.ed.ac.uk/people/ross/tmp/class.dtd">
382|
383|<class>
384|<class_name name="$class"></class_name>
385|
386|<data>
387|START
388|
389|    return *FILE;
390|}
391|
392|#####
393|

```

```

394|sub get_flow
395|{
396|    #this routine will get the molar flows of components and return
397|    #them in a hash
398|    my ($stream) = @_;
399|
400|    my %flow = ();
401|
402|    while($stream =~ m!<slot\s+name\s*=\s*"Component\s">(.*?)</slot>!igs) {
403|        my $data = $1;
404|
405|        $data =~ m!<attribute\s+name\s*=\s*"component\s">
406|            \d+_ \d+_ (.*?) _ \d+_ \d+_ \.html</attribute>!isx;
407|        my $component = $1;
408|
409|        if($data =~ m!<attribute\s+name\s*=\s*"mol_flow\s">(\d+?)
410|            </attribute>!isx) {
411|            $flow{$component} += $1;
412|        }
413|    }
414|
415|    #for those flows not defined assume zero flow
416|    foreach my $key (%flow)
417|    {
418|        $flow{$key} = ($flow{$key} or 0.0);
419|    }
420|    return %flow;
421|}
422|
423|#####
424|

```

References

- [1] R. E. Allen, editor. *The Concise Oxford Dictionary*. BCA, 1991.
- [2] Aspen Technology. Aspen plus manual, release 9. 1994.
- [3] M. Baentsch, G. Molter, and P. Sturm. Webmake: Integrating distributed software development in a structure enhanced web. In *Proceedings of the Third International World Wide Web Conference*. Elsevier Science, 1995.
- [4] G. H. Ballinger, R. Bañares Alcántara, D. Costello, E. S. Fraga, J. Krabbe, H. Lababidi, D. M. Laing, R. McKinnel, J. W. Ponton, N. Skilling, and M. W. Spenceley. épée: a process engineering software environment. *Computers and Chemical Engineering*, 18(Supplement):S283–S287, 1994.
- [5] G. H. Ballinger, E. S. Fraga, and R. Bañares Alcántara. A case study in process engineering tool integration. *Computers and Chemical Engineering*, 19(Supplement):S759–S764, 1995.
- [6] B. Bayer, R. Schneider, and W. Marquardt. Integration of data models for process design - first steps and experiences. *Computers and Chemical Engineering*, 24:599–605, 2000.
- [7] R. Bentley, T. Horstmann, K. Sikkell, and J. Trevor. Supporting collaborative information sharing with the world wide web: The BSCW shared workspace system. In *Proceedings of the Fourth International World Wide Web Conference*. O'Reilly & Associates, 1995.
- [8] T. Berners-Lee. Information management: A proposal. <http://www.w3.org/History/1989/proposal.html>, 1989.
- [9] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. RFC 2396 uniform resource identifiers (URI): Generic syntax. <http://www.ietf.org/rfc/rfc2396.txt>, 1998.
- [10] T. Berners-Lee and R. Cailliau. WorldWideWeb: Proposal for a hypertext project. <http://www.w3.org/Proposal.html>, 1990.
- [11] R. Bogusch, B. Lohmann, and W. Marquardt. Computer-aided process modelling with MODKIT. *Computers and Chemical Engineering*, 25:963–995, 2001.
- [12] M. H. Brown and M. A. Najork. Distributed active objects. In *Proceedings of the Fifth International World Wide Web Conference*. Elsevier Science, 1996.

- [13] T. Budd. *An introduction to object-oriented programming*. Addison-Wesley, 2nd edition, 1998.
- [14] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. Pagespace: An architecture to coordinate distributed applications on the web. In *Proceedings of the Fifth International World Wide Web Conference*. Elsevier Science, 1996.
- [15] D. J. Costello, E. S. Fraga, N. Skilling, R. Bañares Alcántara, J. Krabbe, D. M. Laing, R. C. McKinnel, J. W. Ponton, and M. W. Spenceley. épée: A support environment for process engineering software. *Computers and Chemical Engineering*, 20(12):1399–1412, 1996.
- [16] R. Coyne, M. Ehrenberger, and the n -dim Group. Information modelling for software engineering: An illustrative project history. Technical Report EDRC 05-88-94, Engineering Design and Research Centre, Carnegie Mellon University, 1994.
- [17] A. Dix. Challenges and perspectives for cooperative work on the web. In *Proceedings of the ERCIM workshop on CSCW and the Web*. Arbeitspapiere der GMD 984, GMD, 1996.
- [18] J. M. Douglas. *Conceptual Design of Chemical Processes*. McGraw-Hill, 1988.
- [19] Federal Networking Council. Definition of internet. <http://www.fnc.gov/Internet.res.html>, 1995.
- [20] E. S. Fraga, G. Ballinger, J. Krabbe, D. M. Laing, R. C. McKinnel, J. W. Ponton, N. Skilling, and M. W. Spenceley. The implementation of a portable object oriented distributed process engineering environment. Technical Report 1994-17, University of Edinburgh, Chemical Engineering Department, 1994.
- [21] E. S. Fraga and D. M. Laing. Surf's up. *The Chemical Engineer*, (593):15–16, 27 Jul 1995.
- [22] N. Freed and N. Borenstein. RFC 2046 Multipurpose Internet Mail Extensions (MIME) part two: Media types. <http://www.ietf.org/rfc/rfc2046.txt>, 1996.
- [23] J.-J. Giradot, M. Beigbeder, and P. Jaillon. W3 as the ultimate integration tool. <http://orgwis.gmd.de/W4G/WS95/postions/Giradot.html>, 1995.
- [24] S. Gonnet, G. Mannarino, H. Leone, and G. Henning. A task and version-oriented framework for modelling and managing the process design process. In M. F. Mallone, J.A. Trainham, and B. Carnahan, editors, *Foundations of Computer Aided Process Design*, pages 450–453. AIChE, 1999.
- [25] A. Hallet. 3D web integrated flowsheet viewer. Master's thesis, University of Edinburgh, 1996.
- [26] J. W. Hawtin and P. W. H. Chung. Concurrent engineering system for supporting step based activity model. *Computers and Chemical Engineering*, 22(Supplement):S781–S784, 1998.
- [27] G.Q. Huang and K.L. Mak. Issues in the development and implementation of web applications for product design and manufacture. *International Journal of Computer Integrated Manufacturing*, 14:125–135, 2001.

- [28] G.Q. Huang and K.L. Mak. Web-integrated manufacturing: recent developments and emerging issues. *International Journal of Computer Integrated Manufacturing*, 14(1):3-13, 2001.
- [29] J. I. Kroshwitz and M. Howe-Grant, editors. *Kirk-Othmer Encyclopedia of Chemical Technology*, volume 13. John Wiley and Sons, 4th edition, 1994.
- [30] D. Leal. ISO 10303-221, Application Protocol: Functional data and their schematic representation for process plant. June 1996.
- [31] S. Levy, E. Subrahmanian, S. Konda, R. Coyne, A. W. Westerberg, and Y. Reich. An overview of the n -dim environment. Technical Report EDRC 05-65-93, Engineering Design and Research Center, Carnegie Mellon University, 1993.
- [32] M. L. Lu, R. Batres, and Y. Naka. A hypermanager for a computer integrated concurrent process engineering environment. *Computers and Chemical Engineering*, 20(Supplement):S1155-S1159, 1996.
- [33] M. L. Lu, Y. Naka, K. Shibao, X. Z. Wang, and C. McGreavy. A multi-dimensional object-oriented information model for chemical engineering. In *CE95*. Concurrent Technologies Corporation, 1995.
- [34] M. L. Lu, Z. Z. Shi, X. Z. Wang, C. McGreavy, and Y. Naka. Agent configuration in a concurrent engineering environment. In *CEEDA96*. Epic Event Management Publications, 1996.
- [35] J. Meddes. Epée 0.2 programmers guide. Technical Report 1996-06, University of Edinburgh, Department of Chemical Engineering, 1996.
- [36] Microsoft. Microsoft windows technologies - Internet Explorer. <http://www.microsoft.com/windows/ie/>.
- [37] Microsoft. Vbscript. <http://msdn.microsoft.com/scripting/vbscript/default.htm>.
- [38] NCSA http Development Team. The common gateway interface. <http://hoochoo.ncsa.uiuc.edu/cgi/overview.html>.
- [39] Netscape Communications. Javascript documentation. <http://developer.netscape.com/docs/manuals/javascript.html>.
- [40] Netscape Communications. Netscape browser documentation. <http://developer.netscape.com/docs/manuals/browser.html>.
- [41] M. Palmer. ISO 10303-231 Application Protocol: Process engineering data: Process design and process specification of major equipment. November 1998.
- [42] J. L. Robertson, E. Subrahmanian, M. E. Thomas, and A. W. Westerberg. Management of the design process: the impact of information modelling. In *PSE94*. Korean Institute of Chemical Engineers, 1994.
- [43] J. Skeels. ISO 10303-227, Application Protocol: Plant spacial configuration. 1997.

- [44] G. Stephanopoulos, J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis, and C. Siletti. Design kit: An object-oriented environment for process engineering. *Computers and Chemical Engineering*, 11(6):655–674, 1987.
- [45] E. Subrahmanian, S. Konda, S. Levy, Y. Reich, A. W. Westerberg, and I. Monarch. Equations aren't enough: Information modelling in design. Technical Report EDRC 05-77-93, Engineering Design and Research Centre, Carnegie Mellon University, 1993.
- [46] Sun Microsystems. Java technology. <http://www.sun.com/java/>.
- [47] T. L. Teague and J. T. Baldwin. The emerging discipline of chemical engineering info transfer. In *Foundations of computer aided process design*. AIChE, 1999.
- [48] W.F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15:637–654, 1985.
- [49] J. Trevor, R. Bentley, and G. Wildgruber. Exorcising daemons: A modular and lightweight approach to deploying applications on the web. In *Proceedings of the Fifth International World Wide Web Conference*. Elsevier Science, 1996.
- [50] W3C Consortium. XML home page. <http://www.w3c.org/xml/>, 2000.
- [51] W3C Consortium. HTML home page. <http://www.w3.org/MarkUp/>, 2001.
- [52] H. Wahyu. *Identification of side reactions and byproducts in process synthesis*. PhD thesis, University of Edinburgh, 1999.
- [53] N. Waletzko and L. D. Schmidt. Modeling catalytic gauze reactors: HCN synthesis. *AIChE Journal*, 34(7):1146–1155, July 1988.
- [54] A. W. Westerberg, E. Subrahmanian, Y. Reich, S. Konda, and the n -dim group. Designing the process design process. *Computers and Chemical Engineering*, Supplement:S1–S9, 1997.
- [55] A. W. Westerberg and the n -dim Group. Distributed and collaborative computer-aided environments in process engineering design. In *Intelligent Systems in Process Engineering*. AIChE, 1995.
- [56] T. K. Woo and M. J. Rees. A synchronous collaboration tool for the world wide web. In *Proceedings of the Second World Wide Web Conference*. Published electronically at http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/WWW2_Proceedings.html, 1994.
- [57] World Wide Web Consortium. W3c home page. <http://www.w3.org/Consortium/>.