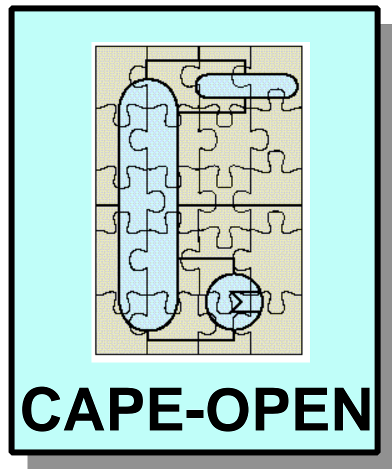
CAPE-OPEN Open Interface Specifications

Unit Operations



UNIT Work Package

CO-CUNIT-1 Version 2.0

IMPORTANT NOTICES

Disclaimer of Warranty

CAPE-OPEN documents and publications include software in the form of *sample code*. Any such software described or provided by CAPE-OPEN --- in whatever form --- is provided "as-is" without warranty of any kind. CAPE-OPEN and its partners and suppliers disclaim any warranties including without limitation an implied warrant or fitness for a particular purpose. The entire risk arising out of the use or performance of any sample code --- or any other software described by the CAPE-OPEN project --- remains with you.

Copyright © 1999 CAPE-OPEN and project partners and/or suppliers. All rights are reserved unless specifically stated otherwise.

CAPE-OPEN is a collaborative research project established under BE 3512 "Industrial and Materials Technologies" (Brite-EuRam III), reference BRPR-CT96-0293.

Trademark Usage

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in CAPE-OPEN publications, and the authors are aware of a trademark claim, the designations have been printed in caps or initial caps.

Microsoft, Microsoft Word, Visual Basic, Visual Basic for Applications, Internet Explorer, Windows and Windows NT are registered trademarks and ActiveX is a trademark of Microsoft Corporation.

Netscape Navigator is a registered trademark of Netscape Corporation.

Adobe Acrobat is a registered trademark of Adobe Corporation.

Visio is a registered trademark of Visio Corporation.

Summary

This document, which was produced by the CAPE-OPEN Unit Operations work package, describes the Interface Specifications for the Unit Operations component of the CAPE-OPEN Interface System. It also describes the demonstration implementation of these interfaces in a steady-state, sequential modular simulator. This corresponds to the GRP1 sub-task described in the CAPE-OPEN work plan[1]. Treatment of sub-task GRP2, which deals with steady-state and dynamic equation-oriented simulation, is also included, although this has not yet been prototyped.

This revision of the document contains changes reflecting the experience gained by the Interoperability Task Force of Work Package 4 of Global CAPE-OPEN and describes the changes made at the 0-9-3 revision of the specification.

The document starts with a text description of the requirements identified for an open unit operation component. This is then expressed in Unified Modelling Language and developed into a specification of the interfaces necessary for a CAPE-OPEN unit operations component to plug into a compliant flowsheet simulator. These specifications are provided in both COM and CORBA IDL. There follows an example that shows how the prototype UNIT component, a mixer/splitter, was implemented and a discussion of the issues that arose in the development of the specification and initial analysis of the prototypes. The final part of the document deals with the extension of these interfaces to handle open steady-state and dynamic equation-oriented simulation.

CAPE-OPEN Archival Information

Reference	CO-CUNIT-1 Version 4.0
Coordinated by	BP, DuPont, GCO ITF, GCO M&T group
Date	11th July, 2001
Number of Pages	216
Version	Version 4.0
Filename	CO Unit Operations v4.doc
Developmental Editor(s)	Peter Banks, BP Peter Edwards, DuPont Juan Carlos Rodriguez, DuPont Richard Martin, QuantiSci Mike Williams, QuantiSci Sergio Cebollero, Hyprotech Michael Halloran, AspenTech
Copy Editor(s)	Sergi Sama, Hyprotech Christophe Poulain, Aspentech Jörg Köller, RWTH-I5 Bill Johns, QuantiSci Bertrand Braunschweig, IFP Daniel Pinol, Hyprotech
Proofreading Editor(s)	
Contributor(s)	Costas Pantelides, Imperial College Ben Keeping, Imperial College Lars von Wedel, RWTH, Aachen Michael White, AspenTech Nii Asante, QuantiSci Christian Kuhlmann, QuantiSci David Smith, DuPont Malcolm Woodman, BP Ron Chartres, BP

CAPE-OPEN Document Roadmap

This document is intended primarily for software engineers, who are interested in producing CAPE-OPEN compliant unit operations components.

All other readers need not go beyond **Section 2 Requirements.**

Acknowledgements

The authors of this document wish to acknowledge the particular contribution of the various people whose names appear in the archival section, as well as the more general contribution and encouragement from many other colleagues. Without their help, thoughts and technical expertise, we would not have been able to complete this project. We would also like to thank the companies and organisations involved for supporting the project and providing the significant resources needed to carry out the work.

Contents

1 INTRODUCTION....3

2 REQUIREMENTS....4

2.1 USER REQUIREMENTS FOR AN OPEN FLOWSHEET UNIT COMPONENT....5

2.1.1 Setting the Scene....5
2.1.2 Architecture....7
2.1.3 CO Objects Present in a Compliant Simulator....8
2.1.4 Communications between a UO and a Simulator....9
2.1.5 Scenarios for Unit Data Handling....10
2.1.6 Conceptual Examples....13
2.1.7 Notes on Conceptual Operation....17
2.1.8 Desirable Characteristics for the CO Interface....19
2.1.9 Thermodynamic Properties....19
2.1.10 Numerical Solvers....20
2.1.11 Granularity....21
2.1.12 UO Prototype Scope (Steady-state & Dynamic)....21

2.2 USE CASES AND DIAGRAMS....22

2.2.1 Use Cases Categories....24
2.2.2 Use Cases Priorities....24
2.2.3 Actors....25
2.2.4 Use Cases....26

2.3 SEQUENCE DIAGRAMS....82

2.3.1 Add Unit to Flowsheet (ref. SQ-31-001)....82
2.3.2 Create Unit (ref. SQ-31-002)....83
2.3.3 Edit Unit (ref. SQ-31-003)....84
2.3.4 Evaluate Unit (ref. SQ-31-004)....85
2.3.5 Retrieve Flowsheet (ref. SQ-31-005)....86
2.3.6 Save Flowsheet (ref. SQ-31-006)....87
2.3.7 Set Unit Specific Data (ref. SQ-31-007)....88
2.3.8 Specify Unit's Material Connections (ref. SQ-31-008)....89
2.3.9 Specify Unit's Information Connections (ref. SQ-31-009)....90

3 ANALYSIS....91

3.1 OVERVIEW....92

3.2 INTERFACE DIAGRAMS....93

3.2.1 UNIT Group 1 Interface Diagram (ref. IN-31-001)....93

3.3 SEQUENCE DIAGRAMS....94

3.3.1 Connecting a Port (ref. SQ-31-010)....94
3.3.2 Edit/Viewing Unit's Specific Data (ref. SQ-31-011)....95
3.3.3 Calculate (ref. SQ-31-012)....97

3.4 STATE DIAGRAMS....98

3.4.1 Port State Diagram (ref. ST-31-001 20)....99
3.4.2 Collection of ports and parameters State Diagrams (ref. ST-31-001)....100
3.4.3 Unit Operation State Diagram (ref. ST-31-001 20)....102

3.5 COMPONENT DIAGRAM....103

3.6 INTERFACE DESCRIPTIONS....106

3.6.1 Interface ICapeUnit Methods....108
3.6.2 Interface ICapeUnitEdit Methods....120
3.6.3 Interface ICapeUnitPort Methods....121
3.6.4 Interface ICapeUnitCollection Methods....126
3.6.5 Interface ICapeUnitReport Methods....128

3.6.6 *Interface ICapeIdentification Methods* ... 131
3.6.7 *Interface ICapeUnitPortVariables* ... 131
3.7 SCENARIOS ... 135
3.7.1 *Mixer/Splitter Scenario (ref. SC-31-001)* ... 135

4 INTERFACE SPECIFICATIONS ... 137

4.1 IDL DEFINITIONS ... 138
4.1.1 *COM IDL* ... 138
4.1.2 *CORBA IDL* ... 146

5 NOTES ON ANALYSIS AND INTERFACE SPECIFICATIONS ... 150

5.1 ISSUES TO BE RESOLVED ... 151
5.2 DECISION RATIONALE ... 152
5.2.1 *Whenever a unit is used there must be a check on its validity.* ... 152
5.2.2 *The three types of port are to be merged into one type.* ... 152
5.2.3 *The CO Ports interface will be restricted* ... 152
5.2.4 *Ports have direction* ... 152
5.2.5 *Parameters* ... 153
5.2.6 *User Interfaces* ... 153
5.2.7 *Undo* ... 154
5.2.8 *Persistence* ... 154
5.2.9 *Reporting* ... 156
5.2.10 *Argument Typing* ... 156
5.2.11 *Tracing and Interrupts* ... 157
5.2.12 *Termination* ... 157
5.2.13 *Initialisation* ... 157
5.2.14 *Connecting Streams* ... 158
5.2.15 *Adding and Removing Ports* ... 158
5.2.16 *The Relationship of UNIT with THRM* ... 161

6 PROTOTYPE IMPLEMENTATION ... 164

7 GLOSSARY ... 165

8 REFERENCES ... 166

9 APPENDIX 1: EQUATION-ORIENTED SIMULATION ... 167

9.1 INTRODUCTION ... 168
9.2 AN EQUATION-ORIENTED UNIT OPERATION ... 169
9.2.1 *Equations for a typical EO unit operation* ... 169
9.2.2 *The Equation Set Object (ESO)* ... 169
9.2.3 *Operation of an EO simulator* ... 170
9.2.4 *GRP2 proposal* ... 172
9.2.5 *Interoperability of SM and EO unit operations* ... 172
9.2.6 *Other issues* ... 173
9.2.7 *Extra methods required for UNIT and NUMR* ... 173
9.2.8 *Mixer-splitter Design Scenario For Steady-state EO Unit and Simulator* ... 173
9.2.9 *Extension to Handle EO Dynamic Simulation* ... 176

1 Introduction

This document describes the Interface Specifications for the Unit Operations component of the CAPE-OPEN Interface System. It was produced by the Unit Operations work package (work package 3) of the CAPE-OPEN project and develops the concepts introduced in the CAPE-OPEN Concepts Document[2].

The Unit Operations work package (UNIT) work programme[1] was organised in two parts:

- **GRP1**, which deals with straightforward unit operations using regular fluids in a sequential modular, steady-state simulator. It makes use of the Thermodynamics and Physical Properties (THRM) interfaces developed by work package 2.

- **GRP2**, which extends this scenario to deal with equation-oriented simulators and dynamic simulation. This makes use of both the THRM and the Numerical (NUMR) work package interfaces.

The main body of this document describes the interfaces and prototype demonstrations needed for GRP1, with the extension to GRP2 contained in an appendix, since this scenario has not yet been prototyped.

The UNIT work package itself was organised around a Focus Group and a Review Team. The Focus Group generated proposals, which were then reviewed by the Review Team and iterated until a satisfactory solution was obtained. This process first produced two internal UNIT documents: a Conceptual Requirements document and UML Description document. The latter was an expression of the Conceptual Requirements in Unified Modelling Language, which is the CAPE-OPEN standard for describing and analysing interface specifications. These documents were used to derive the Unit Operations: Interim Interface Specification document, which was the first UNIT deliverable and the basis for the UNIT demonstration prototype. The current document is a synthesis of all of these documents and the learning obtained to date from the prototyping exercise.

The document starts with a text description of the requirements identified for an open unit operation component. This is then expressed in UML and developed into a specification of the interfaces necessary for a CAPE-OPEN unit operations component to plug into a compliant flowsheet simulator. These specifications are provided in both COM and CORBA IDL. There follows an example that shows how the prototype UNIT component, a mixer/splitter, was implemented and a discussion of the issues that arose in the development of the specification. The final part of the document deals with the extension of these interfaces to handle steady-state and dynamic equation-oriented simulation.

This revision of the document contains changes reflecting the experience gained by the Interoperability Task Force of Work Package 4 of Global CAPE-OPEN and describes the changes made at the 0-9-3 revision of the specification.

2 Requirements

2.1 USER REQUIREMENTS FOR AN OPEN FLOWSHEET UNIT COMPONENT
2.2 USE CASES AND DIAGRAMS
2.3 SEQUENCE DIAGRAMS

2.1 User requirements for an Open Flowsheet Unit Component

2.1.1 Setting the Scene
2.1.2 Architecture
2.1.3 CO Objects Present in a Compliant Simulator
2.1.4 Communications between a UO and a Simulator
2.1.5 Scenarios for Unit Data Handling
2.1.6 Conceptual Examples
2.1.7 Notes on Conceptual Operation
2.1.8 Desirable Characteristics for the CO Interface
2.1.9 Thermodynamic Properties
2.1.10 Numerical Solvers
2.1.11 .... Granularity
2.1.12 .... Suggested UO Prototype Scope (Steady-state & Dynamic)

2.1.1 Setting the Scene

Flowsheet simulators are designed to calculate the behaviour of processes. They take a description of the flowsheet topology and process requirements and assemble a model of the flowsheet from a library of unit operations contained in the simulator. For example, the flowsheet below represents a single stage from an oil and gas separation system:

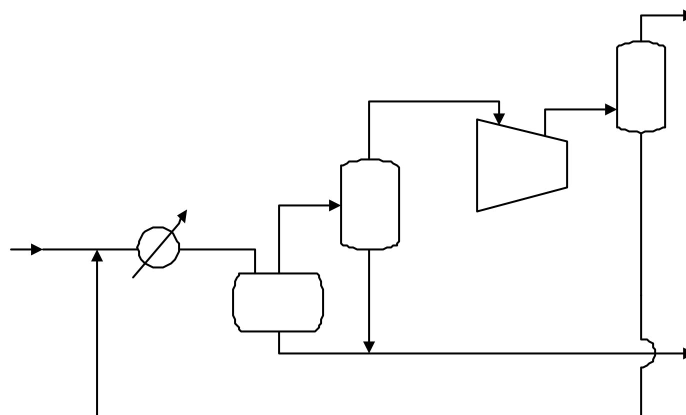


Figure 2.1 Single stage from an oil and gas separation system

This would be simulated in a typical commercial simulator from a description of the topology, probably entered through a GUI and looking much like the above picture, plus a description of the process requirements. It is a simple flowsheet that would use the flash, compressor, cooler and junction unit operations, as shown in figure 2.2:

Although the simulator mimics the plant's behaviour, it is organised differently. For example, in the plant the unit operations are connected together directly, e.g. the cooler connects directly to the separator. In the simulator, the unit operations are not connected to each other, but only to the executive. Also, in this plant there are 3 distinct flash separators, whereas in the simulator there is only one flash algorithm, which is reused by the executive as required by the topology.

This is a straightforward flowsheet that could be adequately simulated by most simulators. However, if, for instance, the separations were to be done with a membrane unit, it might be necessary to use an external representation of the membrane unit to capture the specific performance of a proprietary membrane. Most simulators allow external unit operations to be added, but, because of the proprietary nature of the interface between the unit operations library and the executive and the monolithic structure of the simulator, this is a bespoke activity for each simulator. The result is a non-standard version of the simulator, which can be difficult and expensive to maintain.

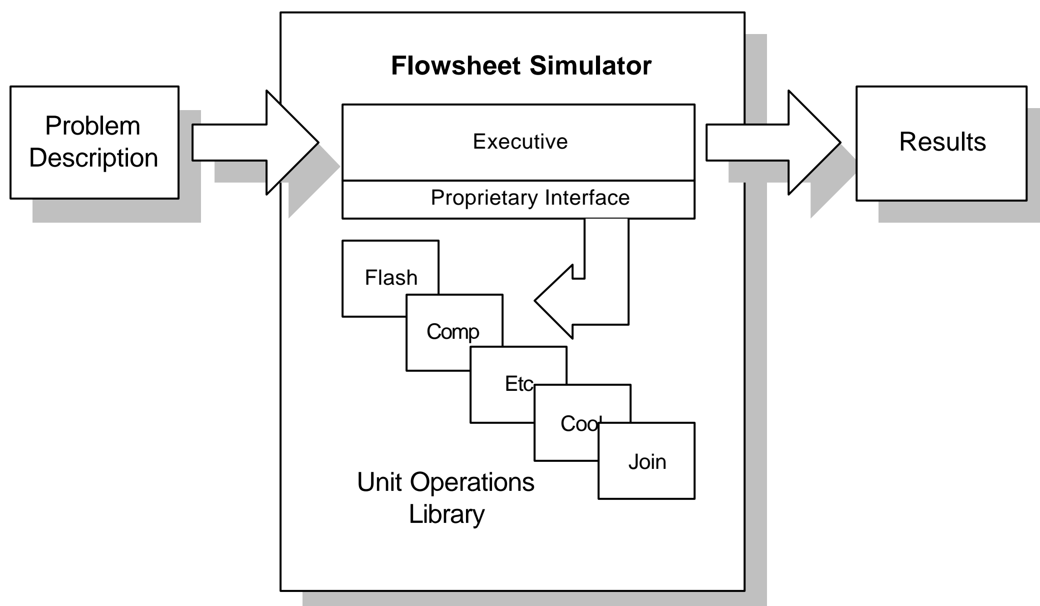


Figure 2.2 Flowsheet simulator schema

The CAPE-OPEN (CO) project envisages a new situation in which a unit operation (and other simulator software sub-systems, such as thermodynamic or numerical packages) can be bought off-the-shelf and plugged directly into any compliant simulator without modification, compiling or linking. It will also continue to work without modification with subsequent versions of the simulator. All that is required is that the unit operation and the simulators conform to the CAPE-OPEN Interface System. This System has been defined by the CO project, which was organised into work packages. Three of these work packages deal with the main sub-systems of a simulator that are likely to be exchanged: unit operations (UNIT), thermodynamics (THRM) and numerical solvers (NUMR). This document is produced by the UNIT work

package and describes the requirements of the part of the CAPE-OPEN Interface System dealing with unit operations.

2.1.2 Architecture

The architecture of the CAPE-OPEN Interface System is based on an object-orientated technology, which allows software systems to be constructed from binary software components. These components are able to talk to each other via defined interfaces. The software components can come from different vendors and may reside on the same machine or be on different machines across a network.

The CAPE-OPEN Interface System is a standard means of connecting an external software component, which models, for example, a unit operation (UO), to any compliant simulator. The Interface can be thought of as a “socket” and “plug”, which exchanges information between the two parts. The simulator and UO do not have to know anything about the internal coding and standards used by the other. The job of the interface is to translate requests for information or action, by either party, into something the other understands.

For example, figures 2.3 and 2.4 show some of the ways in which external facilities can communicate with a host simulator through the interfaces created by the CAPE-OPEN work packages. Please note that these are purely conceptual representations. **The separate connections shown represent the areas of responsibility of the different CO work packages, rather than any physical segregation in the final interface.**

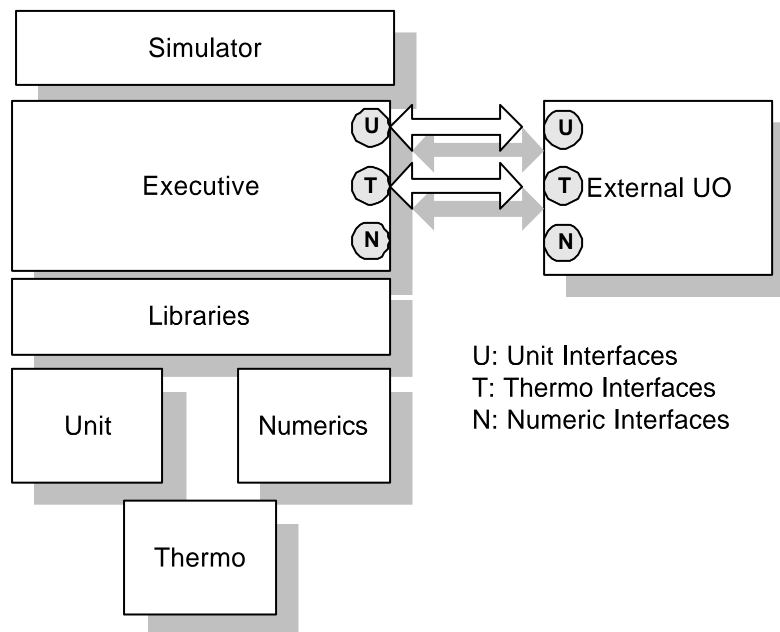


Figure 2.3 Areas of responsibility of the different work packages (approach 1)

The above diagram shows an external UO using the host simulator's thermodynamic facilities. The UO is solving its own equations and so is not shown sending numerical information to the simulator. This is likely to be the default method of operation in sequential modular simulators. An alternative scenario is as follows:

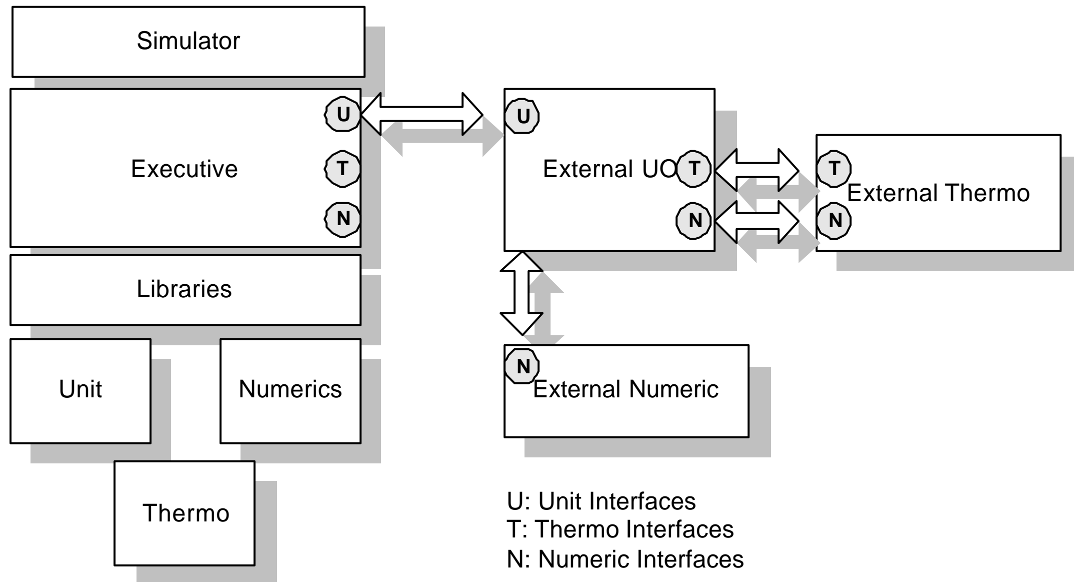


Figure 2.4 Areas of responsibility of the different work packages (approach 2)

This shows an external UO using external thermodynamic and numerical facilities directly. It assumes that the UO requires derivative information from the thermodynamic package, hence the traffic on the part of the interface defined by the NUMR work package. Alternatively, the external thermodynamics package could be plugged into the simulator using the CO Interface. It could still be accessed by the external UO through the CO interface, but would then also be available to all of the unit operations in the simulator's standard library.

2.1.3 CO Objects Present in a Compliant Simulator

In software terms, this architecture requires the following types of objects in a CO system:

- **Unit**: this represents the CO unit and provides methods for initialisation, calculation and reporting. A CO compliant simulator uses these methods to operate the plug-in unit.
- **Simulator**: this provides services that a unit is likely to need from the simulator, such as stream information.
- **Thermo**: this provides physical property services
- **Numerics**: this provides numerical services

The actual location of the services provided by the last two objects may be in separate plug-in software components or may be provided by the simulator itself. As far as the unit is

concerned, it just sees the objects. A unit does not, of course, have to use the simulator's thermo and numerics, if it has them built-in already.

The CO interface system will support the creation and use of these objects. In this way a compliant simulator can use an external unit operation directly.

2.1.4 Communications between a UO and a Simulator

The communications between the UO and the simulator take place during four conceptual phases of a simulation run, using a number of two-way conversations of the following type: (Implementation may differ from simulator to simulator).

Phase 1: Flowsheet creation

In this phase, the user adds a UO to the flowsheet. This causes the simulator to ask the unit how many input and output ports it can handle, to check that the usage is correct.

Table 2.1 Flowsheet Creation

Simulator	External UO
Asks UO how many inputs & outputs it can handle	Responds with number of inputs & outputs

Phase 2: Flowsheet and Unit Initialisation

In this phase, the simulator reads the equipment parameters for each unit in the flowsheet. There are several scenarios, shown in the next section, in which a UO can obtain its data. These differ in how the user inputs the data and where they are stored. In one case, for example, the UO asks the simulator for all of its data. In another, the simulator asks the UO to get its specific data using its own input system, as below.

Table 2.2 Flowsheet and Unit Initialisation

Simulator	External UO
Asks the UO to get its specific data	Obtains data using own input system

Phase 3: Flowsheet Solution and Unit Calculation

In this phase, the simulator evaluates each unit in turn. In the case of CAPE-OPEN UO's, it invokes the UO's "calc" method. The UO in turn invokes methods of the simulator, thermo and numerics objects, as required, during the course of its calculation.

Table 2.3 Flowsheet Solution and Unit Calculation

Simulator	External UO
Invokes calculation	Requests unit feed information from ports
Sends unit feed information	Retrieves specific data from own storage
	Begins calculation
Waits for completion, sending data and providing services as requested	Requests data and services as required during the course of the calculation.
	On completion, sets output ports and returns values to simulator

Phase 4: Flowsheet and Unit Reporting

In this phase, the flowsheet has converged. The simulator asks the UO to provide its results, either as a specific report, or to pass them through the interface for inclusion in a general simulator report. Many units provide their own ASCII text file report which can be output. A simulator may choose to ignore this facility and instead fetch results from the unit and format them itself.

Table 2.4 Flowsheet and Unit Reporting

Simulator	External UO
Asks the UO what output it has	Sends a list of available results
Asks UO to send required results	Sends requested results
Asks UO to output its report	Outputs report

2.1.5 Scenarios for Unit Data Handling

The CAPE-OPEN Interface System does not specify standards for Graphical User Interfaces (GUI's), but it is important to see how the GUI's found on most commercial simulators could fit into the scheme of things. The following diagrams are intended to show possible example scenarios:

Basic: the UO handles its own data storage and retrieval

Intermediate: the UO handles its own data entry and editing, but the simulator handles storage and retrieval

Full: UO data entry and editing performed through the simulator's GUI

Batch – no GUI: UO data included in the simulator batch data file

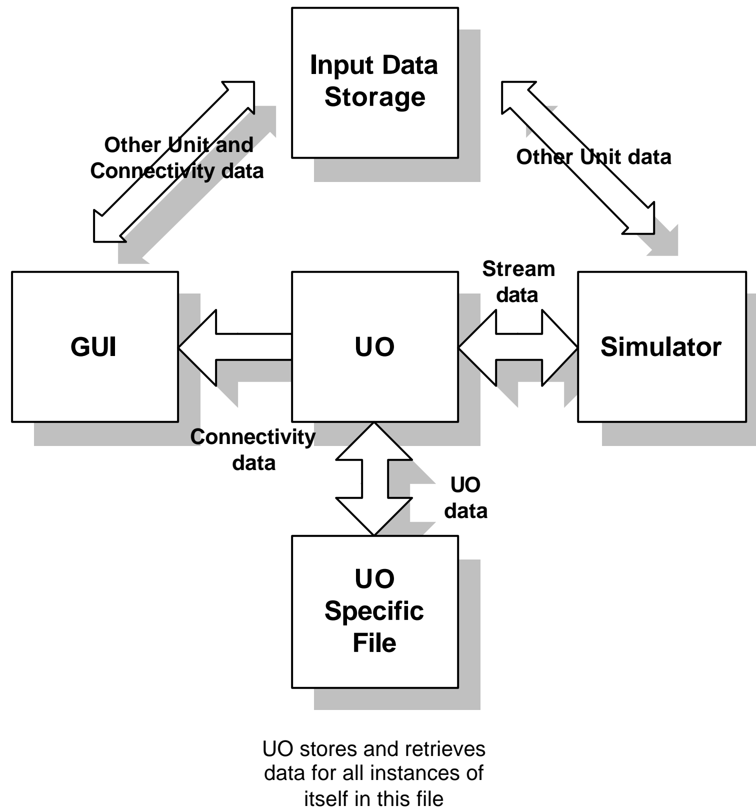


Figure 2.5 Scenarios for Unit Data handling (Basic)

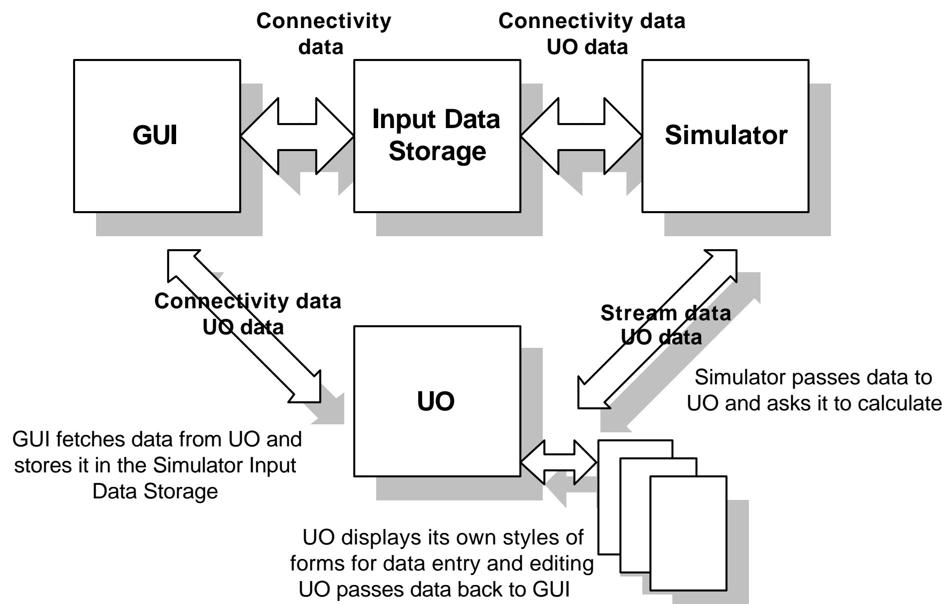


Figure 2.6 Scenarios for Unit Data handling (Intermediate)

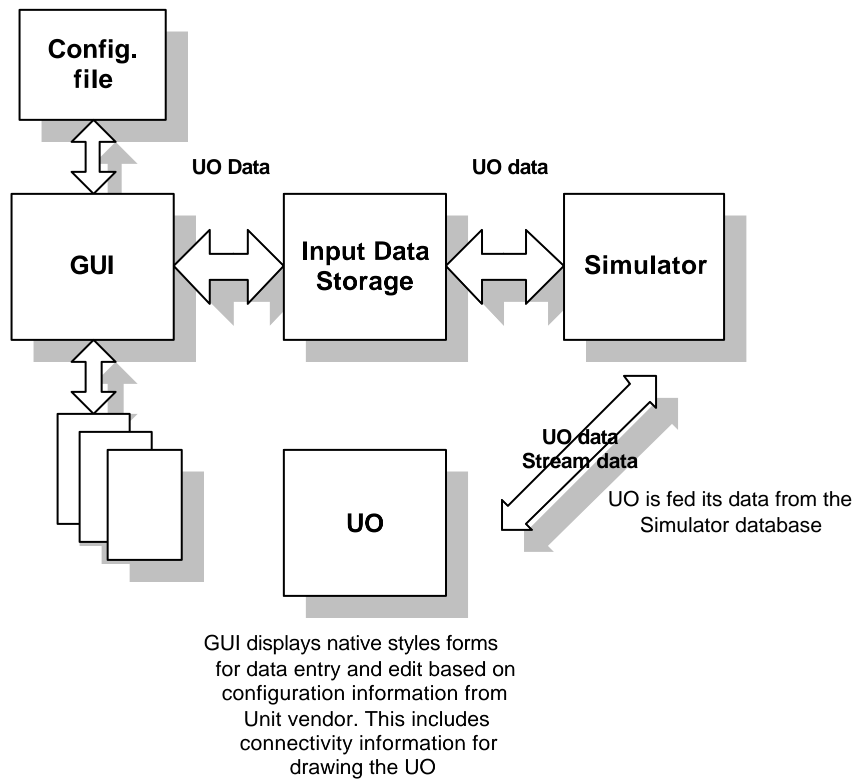


Figure 2.7 Scenarios for Unit Data handling (Full)

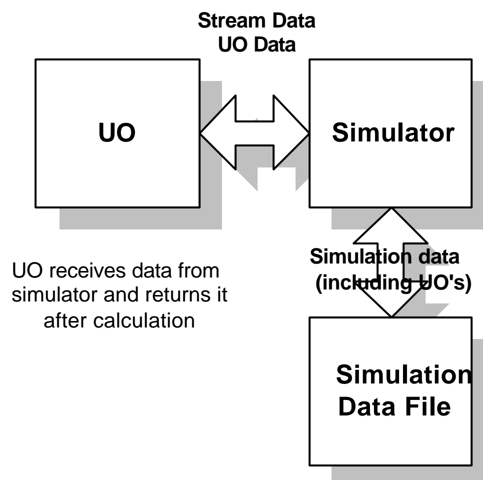


Figure 2.8 Scenarios for Unit Data handling (Batch – No GUI)

2.1.6 Conceptual Examples

These examples show the types of information passing through the interfaces, the way Public Unit Parameters are set and the transfer of stream information between a simulator and an external UO.

Information Traffic through the Interfaces

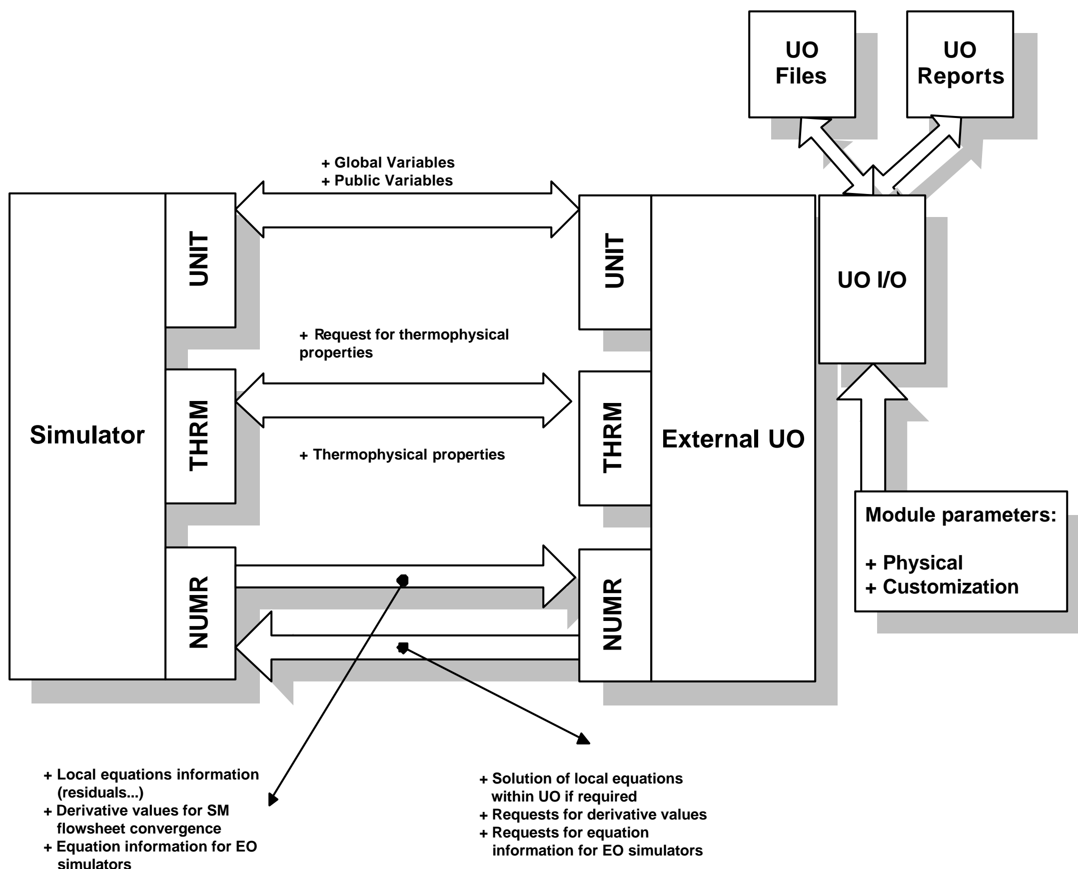


Figure 2.8 Example of Interface Operation

Notes on the above diagram, which shows the types of information passing through the interfaces:

- **Global Variables** - These are essential for the calculation of heat and mass balances, eg stream temperatures, etc. They have mandatory names across all CO interfaces and are included in the CO standards. Although the term “global” has other meanings in programming we use it here to mean that these variables are visible across all CO components.

- **Public Unit Parameters (PUPs)** - These are used for control/optimisation (SM simulators), equation solving (EO simulators) and custom reporting. They are internal variables of an external UO and are named and made accessible to CO interfaces by the UO provider. Their names are not part of the CO standard. PUPs can also be identified as “**read only**”, if they are calculated by the UO and therefore should not be changed by an

optimiser. The UO should raise an error condition, if the simulator attempts to change a read only public parameter.

- **Specific UO data and reports**. These will be handled through I/O routines supplied with the UO, either directly, or by communication with general I/O facilities of the simulator. They are not covered by CO standards. Global data, such as unit feed streams, will be handled through the host simulator's I/O routines. Custom reports created by the host simulator can include information from the external UO via the PUPs defined by the UO provider.

This can be summarised as follows:

- **In a Sequential Modular Simulator (SM)**

  - ✓ The latest values of input global and Public Unit Parameters are requested by and supplied to the UO via the parts of the CO Interface defined by work packages UNIT/THRM .
  - ✓ The UO calculates new values of the output global variables and PUPs and passes them to the host simulator via the parts of the CO Interface defined by work packages UNIT/THRM.
  - ✓ Requests for derivatives at UO solution are passed to the UO via the part of the CO Interface defined by the NUMR work package, if required. However, UO providers do not have to implement derivative calculation to be CO-compliant, although, if they do, their UO will be more versatile and marketable.
  - ✓ Derivative values are passed to the host simulator via the part of the CO Interface defined by the NUMR work package.
  - ✓ During the solution of the UO, repeated thermophysical property requests and responses will occur via the part of the CO Interface defined by the THRM work package.

- **In an Equation Oriented Simulator (EO)**

  - ✓ The required values of global and Public Unit Parameters are passed to and from the UO via the parts of the CO Interface defined by work packages UNIT/THRM .
  - ✓ Calculated values of the UO equation information, eg derivatives and residuals, are passed to the host simulator via the part of the CO Interface defined by the NUMR work package.
  - ✓ During calculation of the UO equation information, repeated thermophysical property requests and responses will occur via the part of the interface defined by the THRM work package.

Resetting Public Unit Parameters

Consider an external UO, which we have assumed to be a distillation column, being manipulated by an optimiser in the host simulator. The optimiser has calculated a new value of 42 for the reflux rate of the column, which it can change, as the UO provider has included a Public Unit Parameter called “reflux_rate” in the list of Public Unit Parameters for this particular UO.

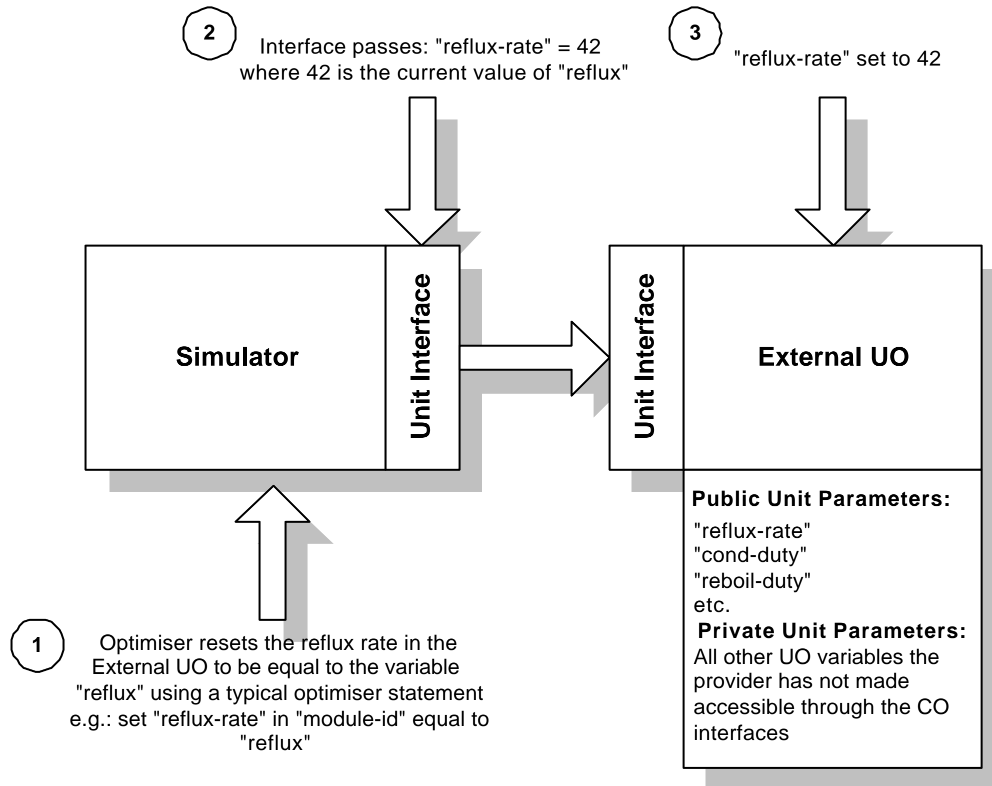


Figure 2.9 Resetting Public Unit Parameters

Note that:

- The Optimiser statement will differ from simulator to simulator, but the PUP name “**reflux-rate**” will be the same on all occasions for this particular UO. It is defined by the UO provider (not CAPE-OPEN). The variables “module-id” and “reflux” represent typical simulator-specific ways of identifying a module and storing a value. They will not be recognised by the CO Interface.

- The form in which the Interface passes the information “reflux-rate” = 42 is the only part of this transaction that is defined by CAPE-OPEN. The external UO will not recognise the simulator variable “reflux”. It will only be passed its current value.

- The list of PUPs will be named by the UO provider, who will choose how many of the UO parameters to expose to the CO Interface and whether they will be read only.

Stream Data Transfer between an SM Simulator and an External UO

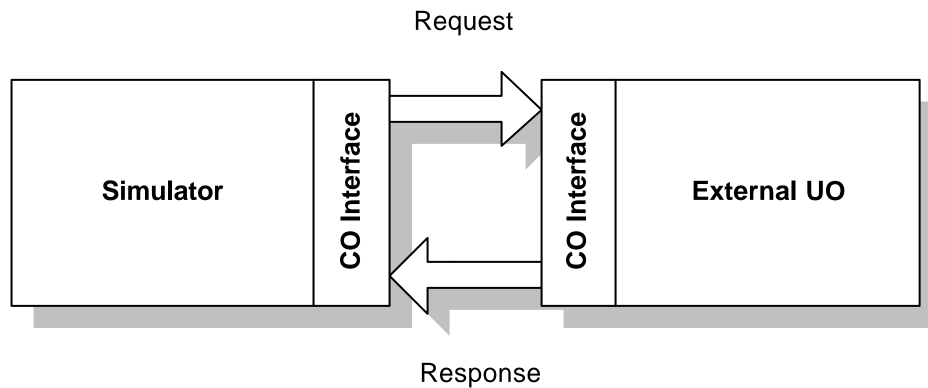


Figure 2.10 Stream Data Transfer between and SM simulator and a Unit Operation

Notes on the above diagram:

- **Request**

UO requests the material stream data it requires from the associated ports:

- ✓ This may be the minimum data, eg T, P and component flows
- ✓ It may be more detailed, eg plus H, S, phase compositions, etc.

- **Response**

The simulator responds in a number of possible ways:

i) If it normally stores the required data items for streams, it returns them and sets the error flag to "OK".

ii) If not, since CAPE-OPEN is only about an interface definition, in a strict implementation the simulator would return the data it could and set the error flag to "caution". It would then be up to the UO provider to decide if the calculation could proceed.

However, we would ***recommend*** that simulator vendors should include with their CAPE-OPEN interfaces the ability to calculate extra stream data, so that the simulator would work with the widest variety of external UO's. If, after calculation, the request is fulfilled, proceed as in (i). If not, proceed as in (ii).

Note - in this diagram we have not separated the parts of the interface defined by the different work packages. As already explained, the diagrammatic separation shown in the previous diagrams was only intended to clarify the areas of responsibility of the work packages.

- **Implications**

  - ✓ CO will define the names of the data items needed to define a stream or port (global variables) and a syntax for carrying out a conversation between an external UO and a simulator.

  - ✓ CO will not define a minimum property set for a stream or port as part of the standard, either for the simulator or the external UO.

  - ✓ The responsibility for calculating material stream data requested by the external UO, but not normally stored by the simulator, lies with the simulator vendor and not with CO.

2.1.7 Notes on Conceptual Operation

- Streams are not part of the UNIT work package, but are part of the overall executive of a simulator and are handled at a project level in CO. There does not need to be a standard definition of streams in CO, so that each simulation package can retain its own stream definitions and still be compliant.

  We have provided a definition of what each stream property is called, so that such information can be labelled when it is passed between software components, e.g. temperature may be called "Temperature". It is this list that needs to be standard, but it also must be maintained so that it can be extended in a systematic way as and when required. (This also applies to the names used for properties). Stream properties include any property required to define the stream, for example polymer molecular weight distributions may need to be included. It may also be useful, for convenience, to define "regular" stream types e.g. one may be defined as mole flows, T, P, H; another in terms in terms of petroleum properties.

  Note that the CAPE-OPEN standards do not refer to streams as such, because streams are uniquely handled in each simulator. Instead, CO refers to ports and thermo objects, which provide a CO view of a stream. A port allows a UO to be connected to thermo objects. Thermo objects pass information in a standard format, performing any necessary conversions between the formats on either side of them (i.e. UO and simulator).

- The handling of error conditions has not yet been fully exploited in the standards. In the first UO prototype demonstration, the implementers agreed to concentrate only on software exceptions, i.e. violations of COM or CORBA. In addition, some of the interface methods are provided with error flags and messages that inform the client on the success or failure of the call. We expect that work on a more refined mechanism will be done as part of Global CAPE-OPEN.

  As an example, we propose that a basic requirement is for the external UO to be able to return to the host simulator either "OK", "failed", or "caution". The condition "caution" could

apply if the external UO has an internal convergence loop that has not fully converged. This non-convergence may not matter if, for example, the UO is in a flowsheet recycle loop and UO convergence is achieved by the time the flowsheet loop has converged.

- Unit Operation Components are also independent of the flowsheet topology. They should only be required to know about their feeds (number, flows etc.) and products. The flowsheet topology is also handled by the Simulator Executive.

- All UO data should be supplied based on the UO's requirements. This could be done by the UO providing the description of the data it requires, e.g. feed stream values, as they are required, in terms of the standard list of stream port properties or regular stream definitions, as in the first point above. On receiving the list, the executive would supply the required values for these data items, or, if it could not, the UO would have to decide if it could continue with the sub-set of data provided. Note that the CO interface will not have standard definitions to handle specific input data for the UO that is only of relevance to that UO, e.g. number of tubes in a heat exchanger. Such data would be supplied via the UO's own interface, or via specifically-configured facilities in the simulator's GUI.

  The process would be repeated to provide calculated data from the UO back to the executive. This system would mean that the CO interface would be completely generic and extensible, and should support both SM and EO simulators.

- The interface should allow the UO to produce a report in its own internal format. Input and calculated variables associated with the UO should also be available to the host simulator, at the discretion of the UO's provider. These would be addressed using a list of variable names invented by the provider. The method of accessing these variables would be a function of the host simulator's scripting language and so would normally differ from simulator to simulator, but the variable names used would be the same in each case. The names are strings that map onto the actual internal variables used by the UO code. A part of the interface has the job of performing this mapping using "get" and "set" routines. The actual variables themselves are not visible.

- It is not the responsibility of the CAPE-OPEN Interface System to ensure overall flowsheet consistency, when an external UO is plugged in. For example, if the UO works internally with a different set of thermodynamic models from the rest of the flowsheet, then the result will not necessarily give a heat balance at the flowsheet level. Obviously, this allows the UO providers to differentiate themselves by providing UO's which ensure consistency, and for the simulation vendors to differentiate themselves by ensuring that results with an external UO are consistent. **In all cases, it is the simulator user's responsibility to ensure that the final system is fit for purpose.** This is an area for further work.

- The ultimate goal is to provide standard interfaces that support connection both at run-time and at the flowsheet creation stage. The first step, however, is to get the former to work. This means that while the user may specify a number of feed streams in the simulator's GUI that is different from that specified through the UO's native interface, this must be picked up at run time. It is a reasonable assumption that a compliant simulator will have a basic icon for user-supplied modules that can handle input and output stream connections.

  The second step would be to provide connections between the UO's native interface and a simulator's GUI, to allow access, for example, to the maximum number of unit feeds and

products supported by the UO and the actual number specified in the GUI. This connection could also provide details of the type of UO, e.g. Pump, to allow access to all the standard GUI icons for a pump. It is not proposed to define a standard form for a GUI icon, as this implies standardisation of GUI's not of interfaces. Obviously the UO provider could supply icons for each individual simulator's GUI, in the form required by the GUI, but this is not included in these standards.

- The work package responsibilities shown on the diagrams above can be summarised as follows:

  - ✓ NUMR part of the interface defines the description of all traffic to do with the solution of the model and flowsheet equations.

  - ✓ THRM part of the interface defines the description of all traffic to do with the provision of thermophysical properties.

  - ✓ UNIT part of the interface defines the description of all traffic to do with the description of the unit and its relationship with the flowsheet, e.g. via its feed and product ports.

2.1.8 Desirable Characteristics for the CO Interface

In the designs that follow, we have tried to meet the following criteria:

- Minimal performance degradation compared to native simulator facilities.

- Minimal impact on the rest of the simulator: other native facilities do not need any change

- Extendable without reworking existing facilities.

- No limitations on the data that can be transferred.

- Consistent design for all the interfaces generated across the project.

- Consistent approach to units of measure conversion across all work packages and with the host simulator.

2.1.9 Thermodynamic Properties

Thermodynamic properties are provided, as far as a CO unit is concerned, by calling methods of a thermo object. These methods provide for setting up a composition and state and calculating various physical properties and derivatives. A unit does not, of course, have to use the thermo object if it has its own thermodynamics system built-in. The services that the thermo object provides may actually come from routines in the simulator itself, or from a separate plug-in software component. The unit does not need to know where they come from, just that the thermo object can provide them. There are a number of ways in which a unit can obtain a thermo object:

- The unit asks the simulator for the current default object. This would usually be the thermo package that is in use in the rest of the flowsheet.

- The user chooses from a list of thermo objects already available and known by the simulator.

- The user chooses from a selection of independent CO compliant thermo objects, if these are available in the operating system registry.

- The unit ignores all thermo objects and performs its own built-in calculations.

The method of access of thermo objects and the description of properties in the CO interface will be determined by the THRM work package. Any derivatives associated with the properties will be described by means determined by the NUMR work package.

2.1.10 Numerical Solvers

The provision of numerical services works in a similar way to thermodynamics, except that, in the case of an SM simulator, it is most likely that a unit will use its own built-in methods. However, even with SM simulators, there can be the need for mathematical information to be transmitted across the CO Interface, as described below.

Simultaneous Modular/Optimisation

In these applications, we require the provision of derivatives of outputs with respect inputs. In a modular simulator there are basically two ways to do this: chain rule or flowsheet perturbation.

- **Chain Rule**

In this approach, derivatives are calculated across a flowsheet loop by calculating the derivatives of all outputs with respect to each input for each module in the loop. These are then chained together to obtain the overall derivatives required. **Thus each UO will need to be able to return these derivatives through the CO Interface in a convenient fashion.**

- **Flowsheet Perturbation**

In this approach, numerical differentiation is used to calculate the overall derivatives for a flowsheet loop by perturbing the input variables individually and evaluating the flowsheet at each perturbation to obtain a numerical approximation of the derivatives required. This is done by the Simulator Executive or the optimisation system. **It imposes no special requirements on the UO and, hence, has no special implications for the CO Interface.**

The method of access to numerical solvers and the description of mathematical properties for the chain rule case above and the general case of EO simulators, will be provided by the NUMR work package.

2.1.11 Granularity

In the first phase of implementation of the Interface, we have restricted granularity to the unit level. This means that in our prototype we have tested plugging a complete UO into a simulator, but not component parts of the UO. For example, we would expect to be able to use the current CO Interface System to plug a complete distillation column UO into a compliant simulator, but not necessarily to plug a new tray hydraulics method into an existing UO. This may be demonstrated in subsequent phases of the Interface.

2.1.12 UO Prototype Scope (Steady-state & Dynamic)

UNIT prototypes have been built around a combined mixer/splitter module, since it represents a simple unit operation that, never-the-less, exhibits the majority of the behaviours of more complex unit operations. This minimises the investment needed to create the unit operation and removes any likelihood of proprietary conflicts. The prototypes reflect the sub-tasks of the work package:

- **Phase 1** interface to a sequential modular simulator
    - ✓ Stage 1 interface to a sequential modular simulator working in steady-state mode without any thermodynamic property required (no thermodynamic interface required).
    - ✓ Stage 2 interface to a sequential modular simulator working in steady-state mode with a list of thermodynamic properties required (thermodynamic interface required).
- **Phase 2** interface extended to include equation oriented simulation, plus a more complex module.

Both phase 1 prototypes were created and interoperability successfully demonstrated by Hyprotech and AspenTech at the ESCAPE 9 meeting in June, 1999, using the Stage 2 interface. The phase 2 prototype was not built before the end of the project.

As a result of work in the Interoperability Task Force of Work Package 4 of Global CAPE-OPEN, this demonstration was repeated by BP at Hyprotech 2000 in November, 2000, using released versions of Aspen Plus 10.2 and HYSYS.Process, modified by patches available from the AspenTech and Hyprotech websites respectively.

2.2 Use Cases and Diagrams

2.2.1 Use Cases Categories
2.2.2 Use Cases Priorities
2.2.3 Actors
2.2.4 Use Cases

Creating a Flowsheet

UC-31-001 Create Unit
UC-31-002 Add Unit to Flowsheet
UC-31-003 Specify Unit's Material Connections
UC-31-004 Specify Unit's Energy Connections
UC-31-005 Specify Unit's Information Connections
UC-31-006 Delete Unit from Flowsheet
UC-31-007 Delete Unit
UC-31-008 Define Unit Report
UC-31-009 Edit Unit
UC-31-010 Save Unit
UC-31-011 Check Unit Specific Data
UC-31-012 Check Unit Information Ports
UC-31-013 Check Public Unit Parameter
UC-31-014 Check Physical Property Methods
UC-31-015 Check Numerical Methods
UC-31-016 Check Material and Energy Ports
UC-31-017 Set Unit Specific Data
UC-31-018 Check Unit
UC-31-019 Get Physical Properties Package
UC-31-020 Get Numerical Package
UC-31-021 Get Public Unit Parameter Names
UC-31-022 Get Unit Specific Data Names
UC-31-023 Save Flowsheet
UC-31-024 Restore Unit
UC-31-025 Retrieve Flowsheet
UC-31-026 Add New Port to Unit
UC-31-027 Delete existing Port

Running a Flowsheet

UC-31-028 Evaluate Unit
UC-31-029 Resume Unit Calculations
UC-31-030 Interrupt Unit Calculations
UC-31-031 Compute Unit Derivatives
UC-31-032 Perturb Unit
UC-31-033 View Interrupted Unit

UC-31-034 Unit Operation Times Out
UC-31-035 Get Input Material Streams from Input Ports
UC-31-036 Get Input Energy Streams from Input Ports
UC-31-037 Get Values of Public Unit Parameters Through Input Ports
UC-31-038 Set Output Material Streams Through Ports
UC-31-039 Set Output Energy Streams Through Ports
UC-31-040 Set Public Unit Parameters On Output Information Ports
UC-31-041 Get Value of Public Unit Parameter
UC-31-042 Set Value of Public Unit Parameter
UC-31-043 Get Stream Data from Input Ports
UC-31-044 Set Stream Data through Output Ports

Looking at Results

UC-31-045 Produce Unit Report

The next sections formalise the description of the user requirements for unit operations interfacing in the CAPE-OPEN Interface System, described in the previous section. They provide a Unified Modelling Language (UML)[3] description of the interfaces, which is the basis for the software design described in later sections.

The first step in the UML process is to express the user requirements in the form of a Use Case Model, which is described in this section. It identifies the "users" of the system, called Actors, and describes, in the form of Use Cases, what they wish the system to do. It also identifies the boundaries of the system, which, in the case of the GRP1 sub-task, is a Unit Operation Model of a steady-state, sequential modular process simulator. Other types of simulator units, such as dynamic or equation oriented, were deferred to GRP2.

In the case of the UNIT GRP1 exercise, the process of identifying and describing the Actors and creating the Use Cases required several iterations to ensure that the user requirements were fully and consistently represented. It also identified the need for consistent names and definitions of the entities involved in flowsheet simulation. From this, a CAPE-OPEN Glossary took shape and was spun off as a separate and extremely useful exercise.

After this, the analysis phase moved on to consider the dynamic nature of the Use Cases with Sequence Diagrams and the static relationships between objects that could be identified from the Use Cases.

The Sequence Diagram is a graphical representation of the Use Case description, showing how the sequence of actions, messages and responses occur over time. For simple Use Cases, a Sequence Diagram is not necessary, but if a Use Case is very complex, with many branches and conditions, or loops then more than one diagram may be needed. By taking this graphical view of the dynamics of the Use Case several things may be identified, such as over complexity of the Use Case, over complex responsibilities of the objects identified in the diagram, or even simply not enough detail in the Use Case.

The first step in identifying the static relationships in the Use Cases was to list all the objects (nouns) contained in the Use Case. Then a screening was applied to remove similar words and replace them by the agreed term from the Glossary. Then a further screening was undertaken

whereby those objects identified which were outside our system boundary were ignored. Then the remaining list was examined by playing back the Use Cases to identify the actions (verbs). Only those behaviours identified as needed by the external clients of the system became part of the final interface.

The rest of this section lists and describes the Actors and Use Cases required for the GRP1 sub-task of the Unit Operations work package and shows their relationships in Use Case Maps. Sequence Diagrams are also shown.

2.2.1 Use Cases Categories

- **Unit Use Cases.** Contains all the Use Cases listed in this document. They are applicable to steady-state Flowsheet Units suitable to be exchanged among sequential-modular and non-sequential-modular simulators of processes.

- **General Purpose Use Cases.** Use Cases that express a software requirement to handle CAPE-OPEN Flowsheet Unit Components. These Use Cases do not have a direct impact on the CAPE-OPEN interfaces, and therefore the requirement does not need to be meet by the CAPE-OPEN interfaces

- **Simulation Context Use Cases.** These are Use Cases that list a sequence of actions, expressed as requirements, so that a CAPE-OPEN Flowsheet Unit is guaranteed to be correctly handled in the different simulation environments. Many times these Use Cases do not have a direct impact on the CAPE-OPEN interfaces, but they represent behavioural requirements on the process simulator side. Many times they also use or extend other more specific Use Cases that do have a direct impact on one or more UNIT Interfaces.

- **Specific Unit Operation Use Cases.** These are Use Cases that represent behaviours of CAPE-OPEN Flowsheet Unit components that do have a direct impact on one or several interfaces.

- **Boundary Use Cases.** These are Use Cases in which a Flowsheet Unit is an actor in other CAPE-OPEN Use Cases different from those corresponding to Flowsheet Units (i.e. THRM or NUMR Use Cases)

- **Use Cases covered by the proposed Mixer/Splitter Scenario.** These are a subset of the Use Cases, selected as high priority Use Cases with the purpose of testing the essential functionality expected from CAPE-OPEN Flowsheet Units. A prototypical Mixer-Splitter operation was selected with this aim.

2.2.2 Use Cases Priorities

- **High.** Essential functionality for a Flowsheet Unit. Functionality without which the operation usability or performance of a Flowsheet Unit might be seriously compromised

- **Medium.** Very desirable functionality that will make Flowsheet Units more usable, transportable or versatile. The essence of a Flowsheet Unit is not compromised by this Use Case, although the usability and acceptance of the component can be.

- **Low.** Desirable functionality that will improve the performance of Flowsheet Units. If this Use Case is not met usability or acceptance can decrease.

2.2.3 Actors

- **Flowsheet Builder.** The person who sets up the flowsheet, the structure of the flowsheet, chooses thermo models and the unit operation models that are in the flowsheet. This person hands over a working flowsheet to the Flowsheet User The Flowsheet Builder can act as a Flowsheet User.

- **Flowsheet User.** The person who uses an existing flowsheet. This person will put new data into the flowsheet, rather than change the structure of the flowsheet.

- **Flowsheet Solver.** A sub system responsible for converging the flowsheet by changing the adjustable variables to meet specified convergence criteria.

  In the modular case, this will be done by iterating the adjustable variables

  In the equation-oriented case, this will be done by performing Newton iteration on a sparse set of non-linear equations

  The function of setting sequencing, nesting of solving sequences and relative convergence limits will be covered in the Use Cases of the Numerical Work Package. The Flowsheet Solver would at some point make use of the sequence of computation of the FlowsheetUnits.

- **Optimiser sub system.** Part of the simulation overall system that is responsible for using an objective function, calculated from the flowsheet, in order to search for an optimum. What is optimised could be one unit or a whole process. The optimiser may use an infeasible path method, in which the optimisation and the flowsheet convergence are carried out simultaneously, or a feasible path method, in which the flowsheet is converged at each iteration of the optimiser.

- **Flowsheet Unit.** A software representation of a physical unit operation, or a non-physical unit such as a controller or optimiser.

- **Unit Manager.** The part of a simulator that provides a list of available Flowsheet Units, allows the instantiation of a Flowsheet Unit and enables it to be placed in the flowsheet.

- **Simulator Executive.** The part of a simulator whose job it is to create, or load, a previously stored flowsheet, solve it and display the results.

- **Reporting sub system.** The part of the executive that reports on the outcome of the calculation of the flowsheet. It reports on the state of the streams and Flowsheet Units involved in the flowsheet. Note: Reporting can be done in different ways. Specific

reporting can be done directly by the Flowsheet Unit on a request from the executive. Overall heat and mass balance reporting is done by passing values from each Flowsheet Unit to the Reporting sub system, which has a generalised report generation capability.

2.2.4 Use Cases

This subsection lists all the Use Cases that are relevant for Group 1 Unit Operations Interface. It also provides links to the methods identified in each Use Case. Each category of Use Cases is introduced by the relevant Use Case Diagram.

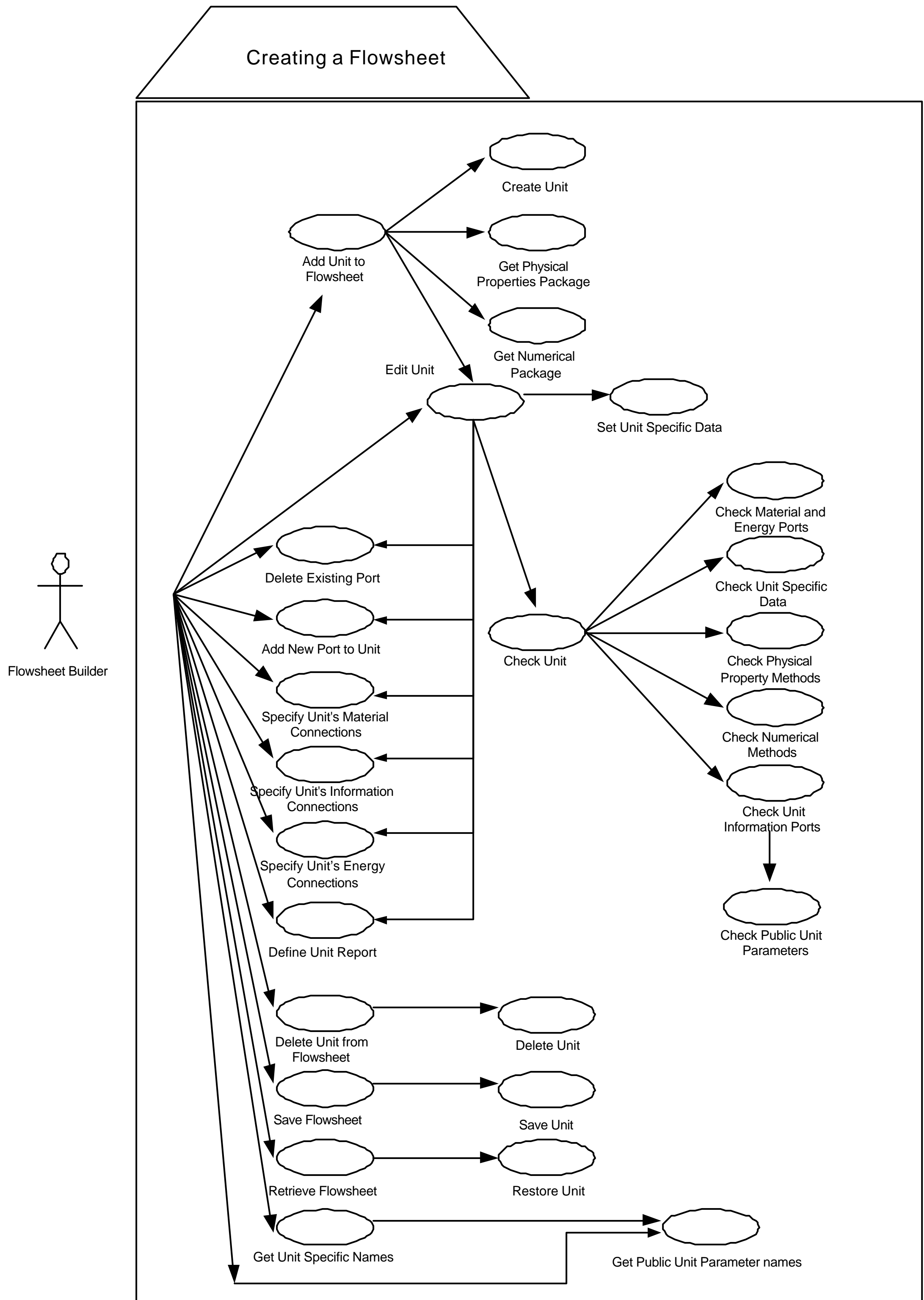


Figure 2.11 Creating a Flowsheet

Create Unit (ref. UC-31-001)

Actors: <Flowsheet Builder>

Priority: <High>

Classification: <Unit Use Case>, <General Use Case>, <Simulation Context Use Case>, <Mixer/Splitter Use Case>.

Status: <A small issue to be resolved (See note)>

<This requirement is fulfilled by mechanism provided by software technology>

Pre-conditions:

- <The selected Flowsheet Unit type may exist>
- <There must be at least one registered Flowsheet Unit of the specified type>

Flow of events:

Basic Path:

Create an instance of a selected Flowsheet Unit type. The Flowsheet Builder gets the list of the available types of Flowsheet Units from the Unit Manager and selects one from the list. The Flowsheet Builder then requests the Unit Manager to create an instance of the selected Flowsheet Unit. The Flowsheet Builder may also supply the location of some specific data to be passed to the Flowsheet Unit. The Unit Manager then creates an instance of the selected Flowsheet Unit. If the creation fails the Unit Manager informs the Flowsheet Builder of the failure. If the creation succeeds the Flowsheet Unit instance is available for use. Note: This does not place the Flowsheet Unit on the flowsheet, this is done by the [Add Unit to Flowsheet] Use Case.

Post-conditions:

- <Creation succeeded>

Exceptions:

- <Creation failed>

See Sequence Diagram [Create Unit]

Add Unit to Flowsheet (ref. UC-31-002)

Actors: <Flowsheet Builder>

Priority: In general <High>, access to Physical and/or Numerical Property Packages might be understood as <Medium> or <Low>

Classification: <Unit Use Case>, <Simulation Context Use Case>, <Mixer/Splitter Use Case>.

Status: <The first part of the Use Case is fulfilled by the normal operation of process simulators>

<Getting access to a Physical Properties Package and/or a Numerical Package is performed through the simulation context the Flowsheet Unit is given> (see SetSimulationContext method)

<May require the use of Initialize>

Pre-conditions:

- <Flowsheet Builder can edit the Flowsheet Unit>

Flow of events:

Basic Path:

Selects a Flowsheet Unit type and adds an instance to the flowsheet. The Flowsheet Builder creates an instance of a Flowsheet Unit using the [Create Unit] Use Case. If the creation fails, the Flowsheet Builder is informed of this, and the Simulator Executive takes no further action. If the creation is successful, the Flowsheet Builder then asks the Simulator Executive to add the Flowsheet Unit instance to the flowsheet. The Simulator Executive adds the Flowsheet Unit to the flowsheet. It may also at this point provide the Flowsheet Unit with access to some default Thermo and Numerical packages. (this may involve the use of [Get Physical Properties Package] and [Get Numerical Package] Use Cases).

If the Flowsheet Unit is successfully added, the Flowsheet Builder may then use the [Edit Unit] Use Case. If the Flowsheet Unit cannot be added to the Flowsheet, the Simulator Executive informs the Flowsheet Builder of this and destroys the Flowsheet Unit. Since it cannot be connected, it is of no use. The same applies if [Edit Unit] fails.

Post-conditions:

- <Flowsheet Unit is successfully initialised (i.e. if Flowsheet Unit needs numerical packages, there is at least one available and if Flowsheet Unit needs thermodynamic packages, there is at least one available>

Exceptions:

- <Initialise Flowsheet Unit fails (i.e. Flowsheet Unit needs numerical or thermo packages and flowsheet can not provide them>

- <Edition of the Flowsheet Unit fails>
- <Failure to present the Flowsheet Unit in the flowsheet>

Subordinate Use Cases:

- [Create Unit]
- [Edit Unit]
- [Get Physical Properties Package]
- [Get Numerical Package]

Specify Unit's Material Connections (ref. UC-31-003)

Actors: <Flowsheet Builder>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>.

Status: <This Use Case is fulfilled by using the following methods: GetPorts, Count, Item, GetPortType, GetDirection, Connect>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

- <There are material streams to connect to the Flowsheet Unit, or they can be created when the connection is to be performed>

- <Flowsheet can present information of material streams to the Flowsheet Unit in the appropriate form (i.e. material objects)>

- <The Flowsheet Unit has material ports to be connected (i.e. input or output)>

Flow of events:

Basic Path:

Flowsheet Unit connections are specified by supplying the connections of its input and output ports.

The Flowsheet Builder asks the Simulator Executive to get the Flowsheet Unit's list of available input and output material ports. The Simulator Executive then asks the Flowsheet Unit for a list of its available input and output material ports, and passes them to the Flowsheet Builder. The Flowsheet Builder also asks the Simulator Executive for a list of available material streams. The Simulator Executive passes the requested list to the Flowsheet Builder.

The Flowsheet Builder selects an input or output port and a material stream from the lists obtained and asks the Simulator Executive to make a connection between the selected stream and the selected port. The Flowsheet Builder may specify a stream name that is different from those supplied. In this case the Simulator Executive will create a new stream with that name. The Simulator Executive asks the Unit to make the requested connection between the selected stream and port. Any previous connection is overwritten. The Flowsheet Builder may repeat this process for the same or other streams and input and output ports. The connection of input and output ports any be performed in any order.

Post-conditions:

- <Connection has been made successfully>

Exceptions:

- <Connection can not be made>
- <Connection can not be overwritten>

Specify Unit's Energy Connections (ref. UC-31-004)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case will be fulfilled by using the following methods: GetPorts, Count, Item, GetPortType, GetDirection, Connect>, <There is no definition of Energy Objects>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <There are energy streams to connect to the Flowsheet Unit, or they can be created when the connection is to be performed (i.e. energy objects)>
- <Flowsheet can present information of energy streams to the Flowsheet Unit in the appropriate form>
- <The Flowsheet Unit has energy ports to be connected (i.e. input or output)>

Flow of events:

Basic Path:

Flowsheet Unit energy connections are specified by supplying the connections of its input and output energy ports.

This is similar to [Specify Unit's Material Connections] except that it is for energy rather than material.

Post-conditions:

- <Connection has been made successfully>

Exceptions:

- <Connection can not be made>
- <Connection can not be overwritten>

Specify Unit's Information Connections (ref. UC-31-005)

Actors: <Flowsheet Builder>

Priority: <Low> if this can be achieved by other means (e.g. using the Flowsheet Unit Parameters)

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: GetPorts, Count, Item, GetPortType, GetDirection, Connect>, <There is no definition of Information Objects>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

- <There are information streams to connect to the Flowsheet Unit, or they can be created when the connection is to be performed>

- <Flowsheet can present information of information streams to the Flowsheet Unit in the appropriate form (i.e. information objects)>

- <The Flowsheet Unit has information ports to be connected (i.e. input or output)>

- <There is at least one other flowsheet object (e.g. Flowsheet Unit) with available information ports. These ports have to be of the opposite direction>

Flow of events:

Basic Path:

The flows of information to/or from other Flowsheet Unit (including optimiser blocks) are specified: The Flowsheet Builder asks the Flowsheet Unit for a list of its available output information ports. A port is selected from the list. The Flowsheet Builder then asks the Simulator Executive to create an information stream and connect it to the selected port. The Simulator Executive creates the new information stream and connects it to the selected port. The connected output port then becomes assigned, and the information stream and the Public Unit Parameter it contains then become available within the flowsheet. Each information streams passes just one Public Unit Parameter. The Flowsheet Builder may repeat this process for the other output ports.

The Flowsheet Builder asks the unit for a list of its available input information ports, and selects one from the list. (Note: selecting an input information port selects a particular variable that the Flowsheet Unit is expecting to receive and use in its calculations. This may be its own or another Flowsheet Unit's "published" Public Unit Parameter(s)) The Flowsheet Builder also gets a list of available information streams from the Simulator Executive, and selects one from the list. He then asks the Simulator Executive to make a connection between the selected information stream and the selected information port. The Simulator Executive makes the requested connection between the selected information stream and port. Any previous connection is overwritten. The Flowsheet Builder may repeat this process for the same or

other information streams and input ports. The connection of input and output ports may be performed in any order.

Post-conditions:

- <Connection has been made successfully>

Exceptions:

- <Connection can not be made>
- <Connection can not be overwritten>

Delete Unit from Flowsheet (ref. UC-31-006)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Simulation Context Use Case>

Status: <Contextual Use Case that is fulfilled by the normal operation of process simulators>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

The Simulator Model Builder selects a Flowsheet Unit and asks for it to be removed from the flowsheet. The Flowsheet Builder selects a Flowsheet Unit in the flowsheet and uses the [Delete Unit] Use Case. The Simulator Executive then removes the Unit from the flowsheet.

Post-conditions:

- <The Flowsheet Unit has been removed from the flowsheet>

Exceptions:

- <The Flowsheet Unit can not be removed from the flowsheet>

Subordinate Use Cases:

- [Delete Unit]

Delete Unit (ref. UC-31-007)

Actors: <Flowsheet Builder>

Priority: <High> deletion of these components is a software requirement anyway

Classification: <Unit Use Case>, <General Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: GetPorts, Count, Item, Disconnect, Terminate>

Pre-conditions:

- <[Create Unit] has been used and successfully passed>

Flow of events:

Basic Path:

This deletes the instance of a Flowsheet Unit.

The Flowsheet Builder requests that the Unit Manager delete the selected Flowsheet Unit instance. The Unit Manager asks the Flowsheet Unit to remove every connection or reference to other flowsheet objects. The Unit Manager then deletes the Flowsheet Unit instance.

Post-conditions:

- <Any connection between the Flowsheet Unit and other flowsheet objects has been removed>
- <Termination proceeded successfully>
- <The Flowsheet Unit has been deleted>

Exceptions:

- <Unit can not be disconnected>
- <Unit cannot be deleted>
- <Termination failed>

Define Unit Report (ref. UC-31-008)

Actors: <Flowsheet Builder>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: GetReports, Get/SetSelectedReport>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit implements report handling>

Flow of events:

Basic Path:

The Flowsheet Builder configures one or more available reports for the Flowsheet Unit based on information made public by the Flowsheet Unit. The Flowsheet Builder asks the Unit Manager to get the list of available report formats from the Flowsheet Unit. If there are some available report formats the Units Manager displays it, so that, the user can select one of them and ask the Flowsheet Unit to set it to be its output format. The Flowsheet Unit sets this format as its output format.

Post-conditions:

- <A report format has been selected>

Edit Unit (ref. UC-31-009)

Actors: <Flowsheet Builder>

Priority: <High>

Status: <This Use Case is fulfilled by using the following methods: the methods listed in the various Use Cases in the Subordinate list below>

Classification: <Unit Use Case>, <Simulation Context Use Case>, <Mixer/Splitter Use Case>.

Pre-conditions:

- <None>

Flow of events:

Basic Path:

The function of this Use Case is to group together Flowsheet Unit parameter editing Use Cases into one as a simulator is likely to have a function to edit Flowsheet Unit data. This Use Case allows any data item to be edited.

The Flowsheet Builder uses the [Set Unit Specific Data], [Specify Unit's Material Connections], [Specify Unit's Information Connections], [Specify Unit's Energy Connections] and [Define Unit Report] Use Cases in any order and any number of times. When the Flowsheet Builder completes his edits, the Unit may use the [Check Unit] Use Case.

Post-conditions:

- <None>

This Use Case only represents a way grouping other Use Cases that commonly will be presented to the Flowsheet Builder as a whole.

Exceptions:

- <Check Unit fails>

Subordinate Use Cases:

- [Set Unit Specific Data]
- [Specify Unit's Material Connections]
- [Specify Unit's Information Connections]
- [Define Unit Report]
- [Add New Port to Unit]

- [Delete Existing Port]
- [Check Unit]

Save Unit (ref. UC-31-010)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: Save>

Pre-conditions:

- <Location for saving Flowsheet Unit data exists>

Flow of events:

Basic Path:

The Unit information, including Unit Specific Data and Unit Results if any, are saved under the specified name and location.

Post-conditions:

- <Save succeeded>

Exceptions:

- <Fails to save>

Check Unit Specific Data (ref. UC-31-011)

Actors: <Flowsheet Unit>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>.

Status: <This Use Case is fulfilled by using the following methods: Validate and GetValStatus>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

This is a check on the validity of the whole Flowsheet Unit specific data. The Flowsheet Unit checks some or all of the items supplied to it as specific data to ensure that they are: within the valid range, consistent with other items of data, plus ANY other test the Flowsheet Unit wishes to perform. The Flowsheet Unit may also use default values where items are missing/inappropriate Finally, the Flowsheet Unit returns the appropriate error or warning messages or codes.

Post-conditions:

- <If Validation is successful GetValStatus must return CAPE_VALID, otherwise CAPE_INVALID>

Exceptions:

- <Parameter invalid>
- <Item missing (this may be a consequence of degrees of freedom)>
- <Item out of range>
- <Variable cannot be used in the specified mode, this will cause the state of the Flowsheet Unit to be set to invalid (i.e. ValStatus = CAPE_INVALID)>

Check Unit Information Ports (ref. UC-31-012)

Actors: <Flowsheet Unit>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Simulation Context Use Case>

Status: <This Use Case is fulfilled by using the following methods: Validate and GetValStatus >

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit operation has information ports to check>

Flow of events:

Basic Path:

This is used by Flowsheet Units, such as optimisers, that need to access the Public Unit Parameters of other Units, both native to the simulator and CAPE-OPEN external units. The Unit asks each of its connected information ports to check its Public Unit Parameters. Each connected information port then asks the Simulator Executive to check that it can set or get the Public Unit Parameters as required (i.e. input ports will ask for a check that they can "get", output ports will ask for a check that they can "set"). The Simulator Executive then calls the information port at the other end of the information connection and asks its Flowsheet Unit (i.e. a target Unit) to use the [Check Public Unit Parameter] Use Case. At the end of checking all connected information ports, the Unit returns a summary of all error messages and/or codes that were generated.

Note: This was not supported in the CAPE-OPEN prototype.

Post-conditions:

- <If Validation is successful the state of the Flowsheet Unit is Valid (i.e. ValStatus = CAPE-VALID), otherwise CAPE_INVALID>

Exceptions:

- <Variable does not exist>
- <Variable cannot be used in the specified mode, this will cause that the state of the Flowsheet Unit is set to invalid (i.e. Valid = FALSE)>

Subordinate Use Cases:

- [Check Public Unit Parameter]

Check Public Unit Parameter (ref. UC-31-013)

Actors: <Flowsheet Unit>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <Partially implemented by the methods in the interface ICapeParameterSpec>

Pre-conditions:

Flow of events:

Basic Path:

The Unit checks that the given variable is recognised and can be got or set as required. If the variable is not recognised, an error message is returned. If the variable is recognised, the variable is checked against the required mode (i.e. get or set). In case the variable can not be used as requested, an error message is returned. Note: This facilitates use by systems/optimisers/controllers etc., which want to use the Public Unit Parameter and have no other means of performing this check ahead of a run-time use of the variable. Mode is Read or Write.

Post-conditions:

Exceptions:

- <Variable not recognised>
- <Variable can not be used as requested >

Check Physical Property Methods (ref. UC-31-014)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>

Status: <Feedback from Validation Work Package is needed to set the status of this Use Case>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

- <Unit operation has access to thermodynamic methods (either through material objects or directly accessing a thermodynamic package>

- <The physical property methods are associated with material definitions>

Flow of events:

Basic Path:

The Flowsheet Unit checks that all the required physical property methods are available using [Unit Performs Thermo Pre-Calculation Check] Use Case. If any property is unavailable an error message is returned.

Post-conditions:

- <If Validation is successful the state of the Flowsheet Unit is Valid (i.e. ValStatus = CAPE-VALID), otherwise CAPE_INVALID>

Exceptions:

- <Method is unavailable, this will cause that the state of the Flowsheet Unit is set to invalid (i.e. Valid = FALSE)>

Subordinate Use Cases:

- [Unit Performs Thermo Pre-Calculation Check]

Extends:

Other requirements:

Check Numerical Methods (ref. UC-31-015)

Actors: <Flowsheet Unit>

Priority: between <Medium> and <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>

Status: <Unknown>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit operation has access to numerical methods>

Flow of events:

Basic Path:

The Flowsheet Unit checks that all the required numerical method are available using [Unit Performs Numeric Pre-Calculation Check] Use Case. If any method is unavailable an error message is returned.

Post-conditions:

- <If Validation is successful the state of the Flowsheet Unit is Valid (i.e. ValStatus = CAPE_VALID), otherwise CAPE_INVALID>

Exceptions:

- <Method is unavailable, this will cause that the state of the Flowsheet Unit is set to invalid (i.e. Valid = FALSE)>

Subordinate Use Cases:

- <[Unit Performs Numeric Pre-Calculation Check], which is a Numeric Use Case and is described in the Numerical Interface Specification>

Check Material and Energy Ports (ref. UC-31-016)

Actors: <Flowsheet Unit>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: Validate and GetValStatus>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit operation has to material and/or energy ports to check>

Flow of events:

Basic Path:

The Flowsheet Unit checks that all essential ports are connected. An essential port is a port that must be present for the Flowsheet Unit to operate e.g. a pump must have one input and one output.

Post-conditions:

- <If Validation is successful the state of the Flowsheet Unit is Valid (i.e. ValStatus = CAPE_VALID), otherwise CAPE_INVALID>

Exceptions:

- <Essential ports are not connected. This will cause the state of the Flowsheet Unit to be set to invalid (i.e. ValStatus = CAPE-INVALID)>

Set Unit Specific Data (ref. UC-31-017)

Actors: <Flowsheet User>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>.

Status: <This Use Case is fulfilled by using the following methods: Edit, GetParameters, Count, Item, Mode, Value, OriginalSource>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit has specific data whose values need to be specified>

Flow of events:

Basic Path:

The Flowsheet User asks the Simulator Executive to start a Flowsheet Unit's user interface. The Simulator Executive then asks the Flowsheet Unit to start its user interface If the Flowsheet Unit does not have a user interface, the Simulator Executive either generates its own user interface, or tells the Flowsheet Builder that no user interface is available. If the Flowsheet Unit has a user interface, the Flowsheet Unit starts it. The Flowsheet Builder then supplies some or all the Flowsheet Unit's specific data through the user interface. When the Flowsheet Builder completes the data input, he terminates the user interface and the Flowsheet Unit's specific data are passed to the Flowsheet Unit. Note that the Simulator Executive is able to use [Get Unit Specific Data Names] to query the Flowsheet Unit for the list of parameters, so that it can construct a user interface for the Flowsheet Unit.

Post-conditions:

- <Unit specific data have been specified>

Exceptions:

- <Unit does not have a user interface and Simulator Executive can not create a user interface>
- <Data values cannot be specified>

Subordinate Use Cases:

- [Get Unit Specific Data Names]

Check Unit (ref. UC-31-018)

Actors: <Flowsheet Unit>

Priority: <High> or <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: Validate and GetValStatus>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

The Simulator Executive may call an external CAPE-OPEN Flowsheet Unit and ask it to use the [Check Unit Specific Data] [Check Material and Energy Ports] [Check Physical Property Methods] [Check Numerical Methods] [Check Unit Information Ports] Use Cases. These checks are provided by the Flowsheet Unit, without knowledge of how or when they may be called.

Post-conditions:

- <If Validation is successful the state of the Flowsheet Unit is Valid (i.e. ValStatus = CAPE_VALID), otherwise CAPE_INVALID>

Exceptions:

- <Validation failed. This will cause that the state of the Flowsheet Unit is set to invalid (i.e. ValStatus = CAPE_INVALID)>

Subordinate Use Cases:

- [Check Unit Specific Data]
- [Check Material and Energy Ports]
- [Check Physical Property Methods]
- [Check Numerical Methods]
- [Check Unit Information Ports]

Get Physical Properties Package (ref. UC-31-019)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>

Status: <In the future it will be fulfilled by the services provided by SetSimulationContext>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit needs access to Physical Property Packages>

Flow of events:

Basic Path:

The Flowsheet Unit gets access to a CO Physical Properties package.

Post-conditions:

- <None> Check Unit will assure that the Unit has been able to get access to the required Property Packages.

Exceptions:

- <No package exists>
- <Requested package not available>

Get Numerical Package (ref. UC-31-020)

Actors: <Flowsheet Unit>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>

Status: <Unknown>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit needs access to Numerical Packages>

Flow of events:

Basic Path:

The Flowsheet Unit obtains access to a Numerical Package.

Post-conditions:

- <None> Check Unit will assure that the Unit has been able to get access to the required Numerical Packages.

Exceptions:

- <No package exists>
- <Requested package not available>

Get Public Unit Parameter Names (ref. UC-31-021)

Actors: <Flowsheet Builder>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>

Status: <This Use Case is fulfilled by using the following methods: GetParameters, Count, Item, Name>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit has Public Unit Parameter >

Flow of events:

Basic Path:

The Flowsheet Builder requests the list of Public Unit Parameter names for the Flowsheet Unit.

Post-conditions:

- < Public Unit Parameternames supplied>

Exceptions:

- <Failed providing names>

Get Unit Specific Data Names (ref. UC-31-022)

Actors: <Flowsheet Builder>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>

Status: <This Use Case is fulfilled by using the following methods: GetParameters, Count, Item, Name, Mode, OriginalSource>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Unit has specific data whose values need to be specified>

Flow of events:

Basic Path:

The Flowsheet Builder requests the list of specific Data Names for the Flowsheet Unit. These names are a subset of the Public Unit Parameters and represents the input data for the Flowsheet Unit. A simulator's user interface system also could use this Use Case to query the Flowsheet Unit for the list of parameters, so that it could construct a user interface for the Flowsheet Unit.

Post-conditions:

- <Unit specific data names retrieved>

Exceptions:

- <Failed providing names>

Extends:

- [Get Public Unit Parameter Names]

Save Flowsheet (ref. UC-31-023)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Simulation Context Use Case>

Status: <This Use Case is fulfilled by the normal operation of process simulators>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

The Flowsheet Builder requests the Simulator Executive to save the Flowsheet. The Simulator Executive saves its native information in its own native format (this is outside CAPE-OPEN). If there are some external Flowsheet Units in the flowsheet, and no location has been specified for saving their information, the Simulator Executive asks the user to supply a name and location for saving the external Flowsheet Unit data.

The Flowsheet Builder provides a name and location, and the Simulator Executive asks the Flowsheet Unit to save its data in the specified location with the specified name. If a name and location has already been specified, the Simulator Executive asks the Flowsheet Unit to save its data in the previously specified location with the previously specified name. The Flowsheet Unit uses the [Save Unit] Use Case to save its data. If the Flowsheet Unit successfully saves itself, the Simulator Executive stores the name and location in its native file. Each CAPE-OPEN Flowsheet Unit has an entry in the native file that stores this information so that the Flowsheet Unit data can be retrieved.

Note 1: If the external Flowsheet Unit is unable to save its data in the simulator's native format, it will prompt the user for the name of a new file, in which it will store its public and private data. In this case, the simulator will be responsible for recording the name of this Flowsheet Unit file along with the rest of flowsheet data.

Note 2: Connectivity data must be persisted by the simulator, including port names.

Post-conditions:

- <Unit data location has been successfully stored with the simulation file>

Exceptions:

- <Failure saving>

Subordinate Use Cases:

- [Save Unit]

Restore Unit (ref. UC-31-024)

Actors: <Flowsheet Unit>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by using the following methods: Restore>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

The Flowsheet Unit restores its specific data from the specified name and location.

Post-conditions:

Exceptions:

- <Data not found>
- <Bad data>

Subordinate Use Cases:

Extends:

Other requirements:

Retrieve Flowsheet (ref. UC-31-025)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Simulation Context Use Case>

Status: <This Use Case is fulfilled by the normal operation of process simulators>

Pre-conditions:

- <The simulation case to be retrieved exists>

Flow of events:

Basic Path:

The Flowsheet Builder asks the Simulator Executive to retrieve a previously stored flowsheet. The Simulator Executive retrieves the native Flowsheet Units in its usual way. For each CAPE-OPEN Flowsheet Unit in the flowsheet, it recovers the Flowsheet Unit type together with the name and location at which the Flowsheet Unit's data is stored. It then asks the Unit Manager to create an instance of the Flowsheet Unit type. It also recovers details of the Flowsheet Unit's stream connections (which are saved in the simulator's native format) and asks each CAPE-OPEN Flowsheet Unit to connect their ports to those specific streams). If the creation is successful, it asks the Flowsheet Unit to retrieve its data from the name and location specified. It does this using the [Restore Unit] Use Case and asks the Flowsheet Unit to validate the information using the [Check Unit] Use Case. If the Flowsheet Unit fails to restore, the Flowsheet Builder is notified.

Post-conditions:

- <Unit operation has been appropriately created and initialised>
- <Unit operation ports have been connected to the appropriate material, energy or information objects>
- <Unit operation has recovered its data>
- <The state of the Flowsheet Unit is Dirty = FALSE and Valid = TRUE>

Exceptions:

- <Failed to retrieve the flowsheet>
- <Failed to restore the Flowsheet Unit data>
- <Failure to connect Flowsheet Unit>
- <Corrupted Flowsheet Unit data>

Subordinate Use Cases:

[Add Unit to Flowsheet]

- [Specify Unit's Material Connections]
- [Specify Unit's Information Connections]
- [Define Unit Report]
- [Add New Port to Unit]
- [Restore Unit]
- [Check Unit]

Add New Port to Unit (ref. UC-31-026)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by the internal Flowsheet Unit behaviour, if the Unit requires it. No interface method is required>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <…>

Flow of events:

Basic Path:

The Flowsheet Builder requests the Simulator Executive to add a port (input or output) to a specified Flowsheet Unit. The Simulator Executive then requests the Flowsheet Unit to make an additional port (input or output as requested) available for use. If the Flowsheet Unit does not allow ports to be added, or if the maximum number of (input or output) ports allowed by the Flowsheet Unit has been reached, the Flowsheet Unit informs the Simulator Executive of this. The Flowsheet Manager then passes this information to the Flowsheet Builder. If the Flowsheet Unit allows ports to be added, and the maximum number of (input or output) ports it allows has not been reached, it makes another (input or output) port available for use.

Post-conditions:

- <…>

Exceptions:

- <Unit does not allow ports to be added>
- <Maximum number of ports already reached>

Delete existing Port (ref. UC-31-027)

Actors: <Flowsheet Builder>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by the internal Flowsheet Unit behaviour, if the Unit requires it. No interface method is required>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

Flow of events:

Basic Path:

The Flowsheet Builder requests the Simulator Executive to delete a port of the specified Flowsheet Unit. The Simulator Executive then requests the Flowsheet Unit to delete the specified port. If the Flowsheet Unit does not allow ports to be deleted, or if the Flowsheet Unit does not allow the selected port to be deleted, the Flowsheet Unit informs the Simulator Executive of this. The Simulator Executive then passes this information to the Flowsheet Builder. If the Flowsheet Unit allows the selected port to be deleted, the Simulator Executive removes it from the list of available ports.

Exceptions:

- <Port cannot be deleted>

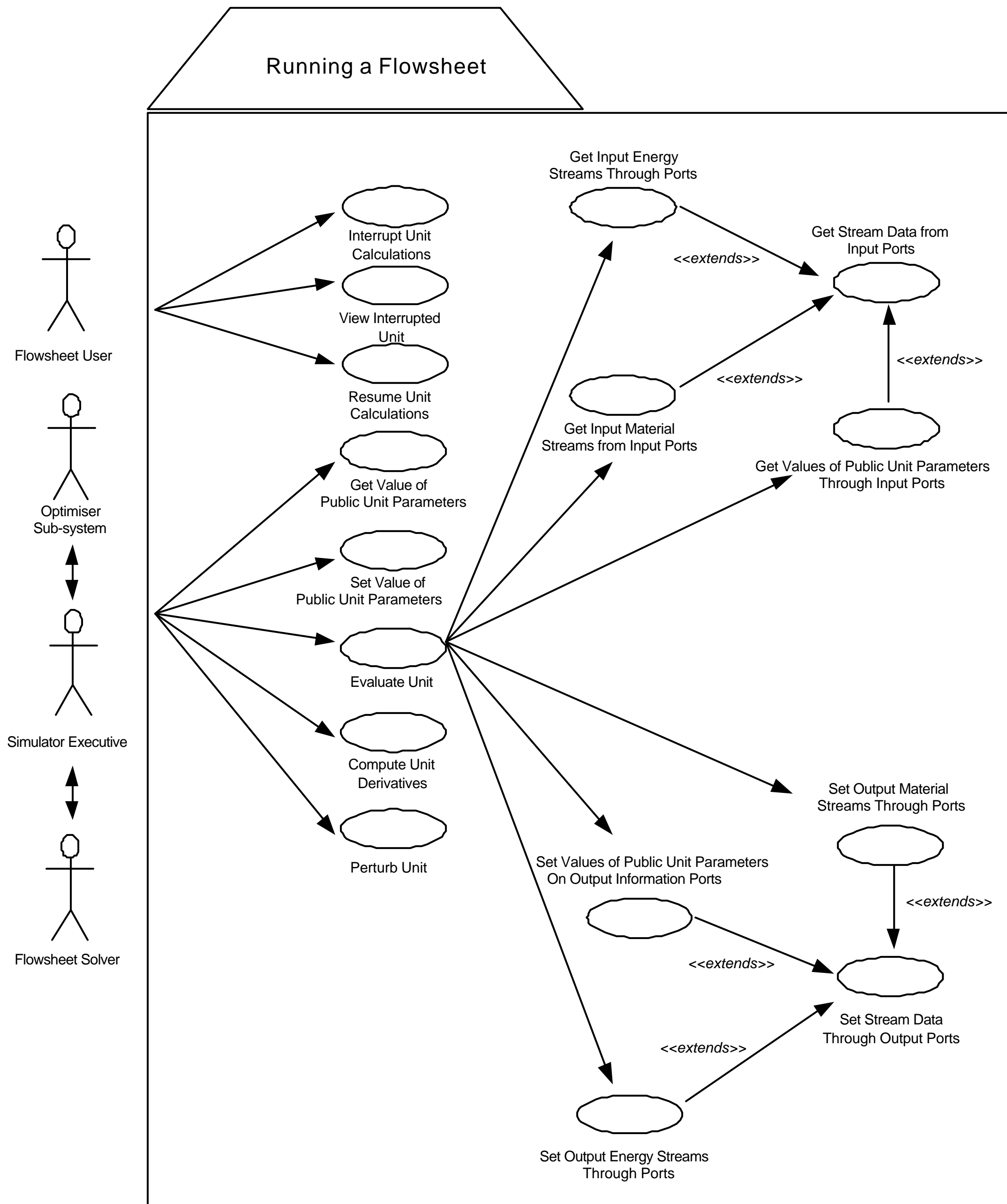


Figure 2.12 Running a Flowsheet

Evaluate Unit (ref. UC-31-028)

Actors: <Flowsheet Solver>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>, <Mixer/Splitter Use Case>

Status: <This Use Case is fulfilled by the following methods: Calculate>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <The input and output ports (material, energy or information) have been connected>

Flow of events:

Basic Path:

The Flowsheet Unit is requested to calculate its required results. The Flowsheet Solver tells the Flowsheet Unit to calculate. The Flowsheet Unit then requests its unit ports to get its input stream data using the [Get Input Material Streams from Input Ports] and [Get Input Energy Streams from Input Ports] Use Cases, as well as the current values of any required Public Unit Parameters (including its own) using the [Get Values of Public Unit Parameters Through Input Ports] Use Case. If some specific data has been provided by the Flowsheet Builder, but has not yet been retrieved by the Flowsheet Unit, the Flowsheet Unit gets this specific data. The Flowsheet Unit then attempts to perform its calculations. It may also make several requests for physical property and numerical services using the {request physical properties} and {request numerical} Use Cases during the evaluation. Upon a successful partial or complete evaluation, the Flowsheet Unit sends the partial or complete definition of its output streams to its output ports using the [Set Output Material Streams Through Ports] and [Set Output Energy Streams Through Ports]. It also updates any changed values of Public Unit Parameters using the [Set Public Unit Parameters On Output Information Ports]. If the Flowsheet Unit cannot calculate, it returns the appropriate error to the Flowsheet Solver.

Post-conditions:

- <The Flowsheet Unit finished its calculations successfully or not>

Exceptions:

- <The Flowsheet Unit may not solve successfully>

Subordinate Use Cases:

- [Thermo: Request Physical Properties]
- [Numerical: ?SolveEquations?]

- [Get Input Material Streams from Input Ports]
- Get Input Energy Streams from Input Ports
- [Get Values of Public Unit Parameters Through Input Ports]
- [Set Public Unit Parameters On Output Information Ports]
- [Set Output Material Streams Through Ports]
- [Set Output Energy Streams Through Ports]

Resume Unit Calculations (ref. UC-31-029)

Actors: <Flowsheet User>

Priority: <Low>

Status: <Not fulfilled. In the future, this will be fulfilled by using the services of the SimulationContext>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <The Flowsheet Unit was interrupted>

Flow of events:

Basic Path:

The Flowsheet Builder requests that an interrupted Flowsheet Unit should resume calculation. It will continue from the set of variable values it has when the resume is initiated. This will probably be used following a [Edit Unit] Use Case from "Creating a Flowsheet"

Post-conditions:

- <The Flowsheet Unit finished its calculations successfully or not>

Exceptions:

- <Cannot resume calculations>

Interrupt Unit Calculations (ref. UC-31-030)

Actors: <Flowsheet User>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <Not fulfilled. In the future this will be fulfilled by using the services of the SimulationContext>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <The Flowsheet Unit has been requested to calculate>

Flow of events:

Basic Path:

Comes from a general interrupt capability. A complementary capability is [Resume Unit Calculations]. Flowsheet Users need to stop a running flowsheet evaluation. They need this interrupt to preserve all values, so that it can restart with the same values of all the calculation variables. The Flowsheet Unit ends up in an interrupted state.

Post-conditions:

- <The Flowsheet Unit has stopped its calculations>

Exceptions:

- <Cannot interrupt the Flowsheet Unit>

Compute Unit Derivatives (ref. UC-31-031)

Actors: <Optimiser sub-system>

Priority: <Medium> Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <Not fulfilled>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <The state of the Flowsheet Unit is Dirty = FALSE and Valid = TRUE>
- <Values are returned by part of the interface, which is provided for derivatives. This is a separate interface to that which returns the output values>
- <We make no prior judgement about when the derivatives are computed>
- <For Flowsheet Units that always compute derivatives, the values will only be supplied to the Optimiser sub-system, if it requests them>

Flow of events:

Basic Path:

The Optimiser sub system requests that the Flowsheet Unit produce derivatives for selected variables with respect to some other selected variable (note that variables may be information or physical values). The Flowsheet Unit supplies the derivatives.

Post-conditions:

- <Derivatives are provided by the Flowsheet Unit>

Exceptions:

- <The Flowsheet Unit cannot compute derivatives>

Perturb Unit (ref. UC-31-032)

Actors: <Optimiser sub-system>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>

Status: <This Use Case is fulfilled by the following methods: Calculate>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <The state of the Flowsheet Unit is Dirty = FALSE and Valid = TRUE>

Flow of events:

Basic Path:

Simply uses [Evaluate Unit] with some perturbed input values

Post-conditions:

- <The Flowsheet Unit finished its calculations successfully or not>

Exceptions:

- <Evaluate fails>

Subordinate Use Cases:

- [Evaluate Unit]

View Interrupted Unit (ref. UC-31-033)

Actors: <Flowsheet User>

Priority: <Medium> because it uses [Edit Unit]

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This Use Case is fulfilled by the following methods: Edit>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Flowsheet Unit has been interrupted>

Flow of events:

Basic Path:

This may be used in combination with [Interrupt Unit Calculations]. A Flowsheet User may want to examine a Flowsheet Unit calculation for various reasons. One reason is to examine the numerical state of its current solution. A Flowsheet User may then choose to change a convergence specification. Another reason may be to change the Public Unit Parameters, e.g. a product purity specification using [Edit Unit].

Post-conditions:

- <Edition finished>

Exceptions:

- <Edition failed>

Unit Operation Times Out (ref. UC-31-034)

Actors: <Flowsheet Solver>

Priority: <Low>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <Not fulfilled>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Flowsheet Unit is calculating>

Flow of events:

Basic Path:

The flowsheet equation solver stops the Flowsheet Unit calculations, when the maximum time limit is exceeded, or when the maximum number of iterations is exceeded. This uses [Interrupt Unit Calculations].

Post-conditions:

- <Flowsheet Unit is in holding mode and it has stopped its calculations>

Exceptions:

- <Flowsheet Unit can not stop its calculations>

Subordinate Use Cases:

- [Interrupt Unit Calculations]

Get Input Material Streams from Input Ports (ref. UC-31-035)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>, <Mixer/Splitter Use Case>

Status: <See UC-31-043>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid material object>

Flow of events:

Basic Path:

As in [Get Stream Data from Input Ports] except: The streams requested are material streams and the Ports are material ports.

Post-conditions:

- <Flowsheet Unit has retrieved data successfully>

Exceptions:

- <Stream data unavailable>

Subordinate Use Cases:

Extends:

- [Get Stream Data from Input Ports]

Get Input Energy Streams from Input Ports (ref. UC-31-036)

Actors: <Flowsheet Unit>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>

Status: <See UC-31-043 >

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid energy object>

Flow of events:

Basic Path:

As in [Get Stream Data from Input Ports] except: The streams requested are energy streams, and the Ports are energy ports.

Post-conditions:

- <Flowsheet Unit has retrieved data successfully>

Exceptions:

- <Stream data may not be available>

Subordinate Use Cases:

Extends:

- [Get Stream Data from Input Ports]

Get Values of Public Unit Parameters Through Input Ports (ref. UC-31-037)

Actors: <Flowsheet Unit>

Priority: <Low> if this can be achieved by other means (e.g. [UC-31-41])

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <See UC-31-043>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid information object>

Flow of events:

Basic Path:

As in [Get Stream Data from Input Ports] except: The streams requested are information streams, and the Ports are information ports, and the stream data consists of Public Unit Parameters. Note: The Public Unit Parameters obtained may include some which belong to the Flowsheet Unit itself as well as some which may belong to another Flowsheet Unit.

Post-conditions:

- <Unit has retrieved data successfully>

Exceptions:

- <Value(s) cannot be found>

Subordinate Use Cases:

Extends:

- [Get Stream Data from Input Ports]

Set Output Material Streams Through Ports (ref. UC-31-038)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>, <Mixer/Splitter Use Case>,

Status: <See UC-31-044>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid material object>

Flow of events:

Basic Path:

As in [Set Stream Data through Output Ports] except: The streams requested are material streams, and the Ports are material ports.

Post-conditions:

- <Values have been set correctly>

Exceptions:

Subordinate Use Cases:

Extends:

- [Set Stream Data through Output Ports]

Set Output Energy Streams Through Ports (ref. UC-31-039)

Actors: <Flowsheet Unit>

Priority: <Medium>

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Boundary Use Case>,

Status: <See UC-31-044>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid energy object>

Flow of events:

Basic Path:

As in [Set Stream Data through Output Ports] except: The streams requested are energy streams, and the Ports are energy ports.

Post-conditions:

- <Values have been set>

Exceptions:

Subordinate Use Cases:

Extends:

- [Set Stream Data through Output Ports]

Set Public Unit Parameters On Output Information Ports (ref. UC-31-040)

Actors: <Flowsheet Unit>

Priority: <Low> if this can be achieved by other means (e.g. [UC-31-42])

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <See UC-31-044>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid information object>

Flow of events:

Basic Path:

As in [Set Stream Data through Output Ports] except: The streams requested are information streams, and the Ports are information ports, and the stream data consists of newly calculated values of one or more Public Unit Parameters. Note: The Public Unit Parameters obtained may include some which belong to the Flowsheet Unit itself as well as some which may belong to another Flowsheet Unit.

Post-conditions:

- <Values have been set correctly>

Exceptions:

Subordinate Use Cases:

Extends:

- [Set Stream Data through Output Ports]

Get Value of Public Unit Parameter (ref. UC-31-041)

Actors: <Simulator Executive>

Priority: <High>

Status: <This requirement is fulfilled by the following methods: GetParameters, Count, Item, Mode, UpperBound, LowerBound, Value>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

- <The Simulator Executive knows the identification used by the Flowsheet Unit for the required variable>

Flow of events:

Basic Path:

The Simulator Executive asks a Flowsheet Unit for the value of one of its Public Unit Parameters by specifying some form of identification for the Public Unit Parameter, which it expects the Flowsheet Unit to recognise. If the Flowsheet Unit recognises the identification, it passes back the value of the requested variable. (Note: The value could be of an unset variable e.g. "EMPTY") If it does not recognise the identification it informs the Simulator Executive that the identification was not recognised. Note: The value could be of an unset variable e.g. "EMPTY".

Post-conditions:

- <Value has been obtained>

Exceptions:

- <Identification not recognised by Flowsheet Unit>

- <Failed retrieving value>

Set Value of Public Unit Parameter (ref. UC-31-042)

Actors: <Simulator Executive>

Priority: <High>

Status: <This requirement is fulfilled by the following methods: GetParameters, Count, Item, Mode, UpperBound, LowerBound, Value>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>

- <The Simulator Executive knows the identification used by the Flowsheet Unit for the required variable>

Flow of events:

Basic Path:

The Simulator Executive passes a variable value and identification to the Flowsheet Unit, and asks the Flowsheet Unit to update the value of its corresponding Public Unit Parameter with the value passed. If the Flowsheet Unit recognises the identification, it attempts to update the recognised variable with the value passed. If it cannot update the variable (e.g. because the value passed is of the wrong type) it informs the Simulator Executive of this. If it does not recognise the identification, it informs the Simulator Executive that the identification was not recognised. Note: This Use Case does not make use of information ports, and it therefore provides an alternative means of accessing a Flowsheet Unit's Public Unit Parameters.

Post-conditions:

- <Value has been set correctly>

Exceptions:

- <Identification not recognised or is incomplete/bad>

- <Cannot update (value of wrong type, value out of range, or variable is read only....)>

Get Stream Data from Input Ports (ref. UC-31-043)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This method is fulfilled by the various methods in the objects connected to ports (i.e. Material, Energy and Information>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid object>

Flow of events:

Basic Path:

The Flowsheet Unit asks each of its input ports to get the current stream data (energy, material or information) from the Simulator Executive.

Each input port then gets the current values of the connected stream data from the Simulator Executive in CAPE-OPEN format. If required, the port converts the stream data into the Flowsheet Unit's native format and passes it back to the Flowsheet Unit.

Post-conditions:

- <Data retrieved>

Exceptions:

- <Failed retrieving data (e.g. data not calculated or initialised)>

Set Stream Data through Output Ports (ref. UC-31-044)

Actors: <Flowsheet Unit>

Priority: <High>

Classification: <Unit Use Case>, <Specific Unit Use Case>

Status: <This method is fulfilled by the various methods in the objects connected to ports (i.e. Material, Energy and Information>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Port is connected to a valid object>

Flow of events:

Basic Path:

The Flowsheet Unit passes its output stream data (information, material or energy) to its output ports (in its own native format) and asks its output ports to update the Simulator Executive's data. The output ports then (if required) perform any necessary conversion of the stream data into CAPE-OPEN format, and pass the output stream data to the Simulator Executive.

Post-conditions:

- <Data set>

Exceptions:

- <Failed setting data>

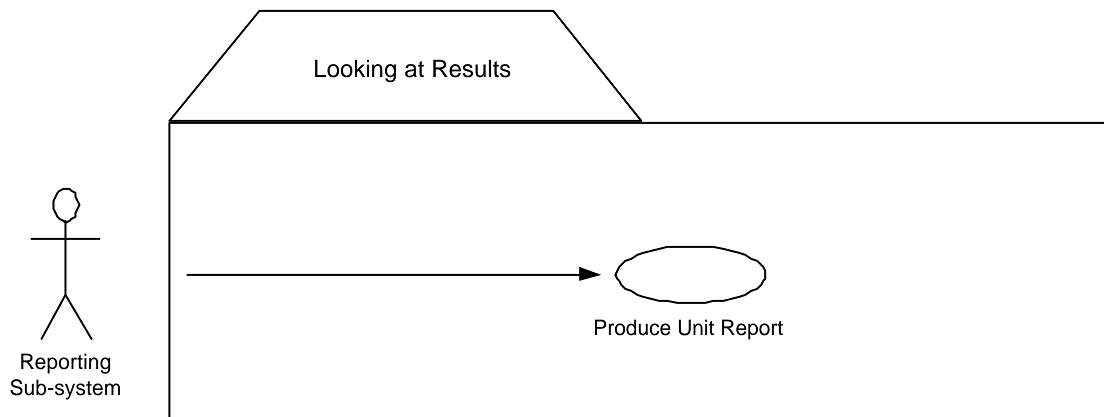


Figure 2.13 Looking at Results

Produce Unit Report (ref. UC-31-045)

Actors: <Reporting sub system>

Priority: <Low> <Medium> provided that the Flowsheet Unit can publish as many variables as it wants, so that Simulator Executive's reporting capabilities can be used

Classification: <Unit Use Case>, <Specific Unit Use Case>, <Mixer/Splitter Use Case>

Status: <This method is fulfilled by the following methods: ProduceReport>

Pre-conditions:

- <[Add Unit to Flowsheet] has been used and successfully passed>
- <Type of report to be produced has been defined>

Flow of events:

Basic Path:

Produce a report on this Flowsheet Unit. The Reporting sub system requests the Flowsheet Unit to produce a report. The Flowsheet Unit produces the report including the information requested by the [Define Unit Report] Use Case. It may also report on the status of the calculation.

Post-conditions:

- <Report has been produced>

Exceptions:

- <The Flowsheet Unit does not produce the report>
- <The Flowsheet Unit cannot produce the report currently>
- <The report is not defined>

2.3 Sequence diagrams

These Analysis Sequence Diagrams are included to illustrate how the definitions of some of the critical methods were achieved and to divide the textual description of the more complex Use Cases into smaller actions that occur in a specific order. To keep them as simple as possible, only exceptions that result in non-trivial behaviour are shown and no returning responses from messages sent from one object to another are included.

2.3.1 Add Unit to Flowsheet (ref. SQ-31-001)

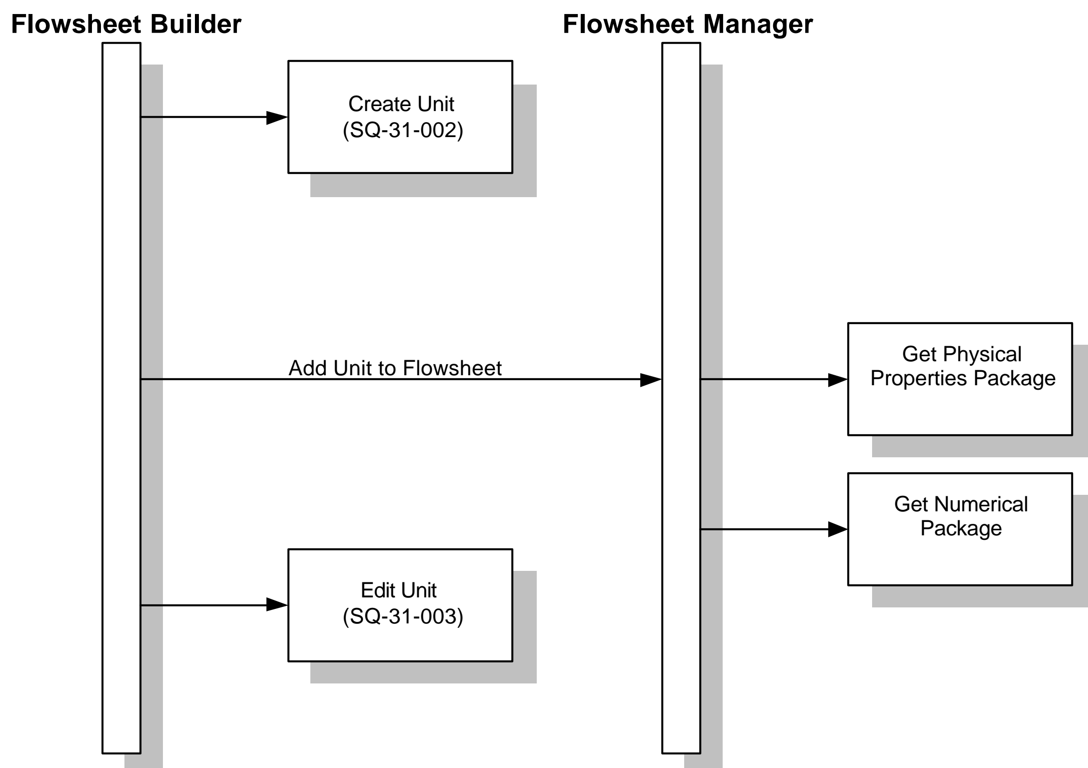


Figure 2.14 SQ – Add Unit to Flowsheet

2.3.2 Create Unit (ref. SQ-31-002)

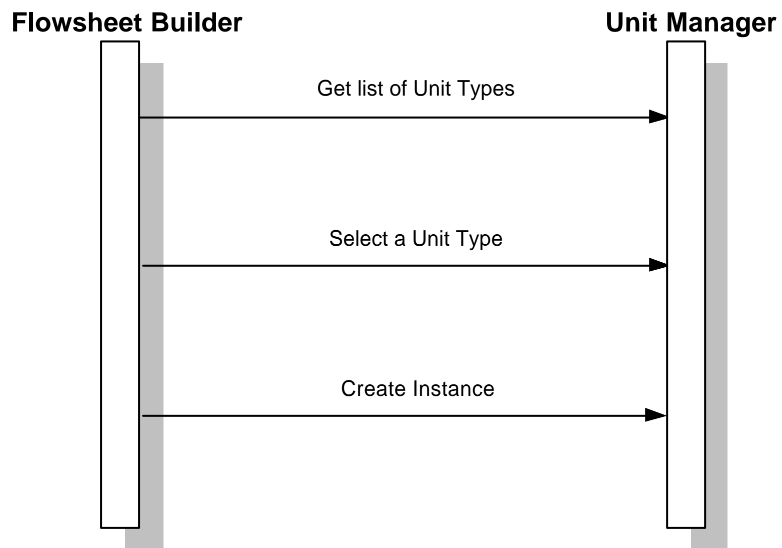


Figure 2.15 SQ – Add Unit to Flowsheet

2.3.3 Edit Unit (ref. SQ-31-003)

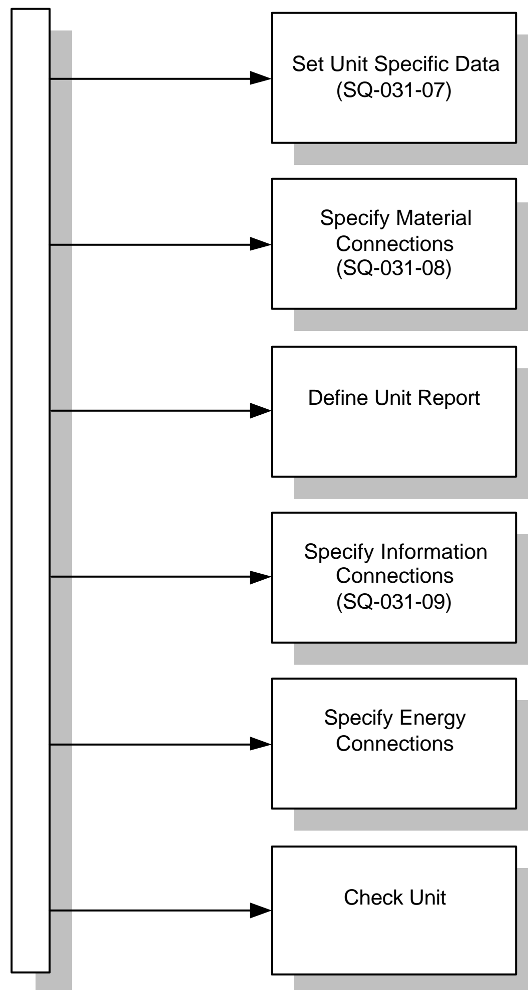


Figure 2.16 SQ – Edit Unit

2.3.4 Evaluate Unit (ref. SQ-31-004)

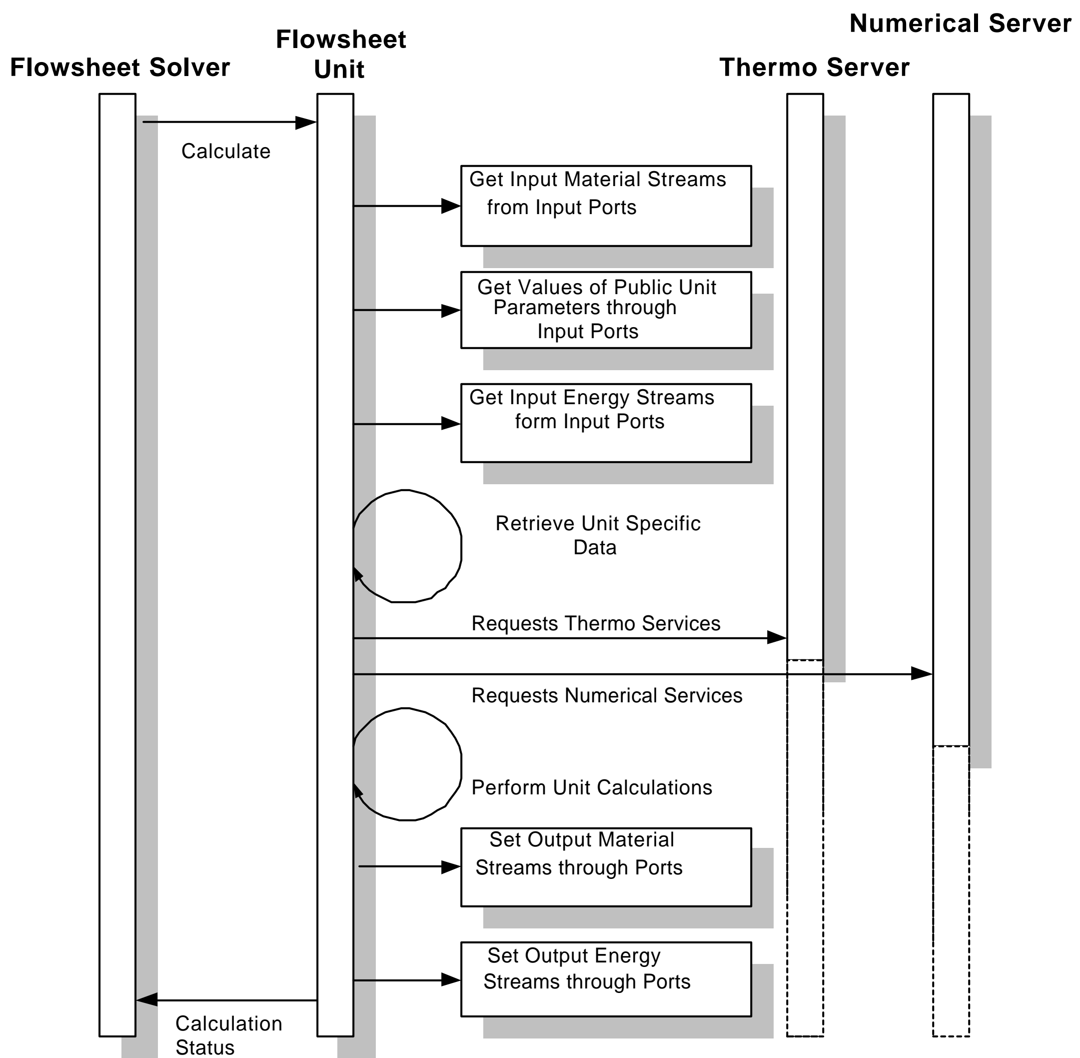


Figure 2.17 SQ – Evaluate Unit

2.3.5 Retrieve Flowsheet (ref. SQ-31-005)

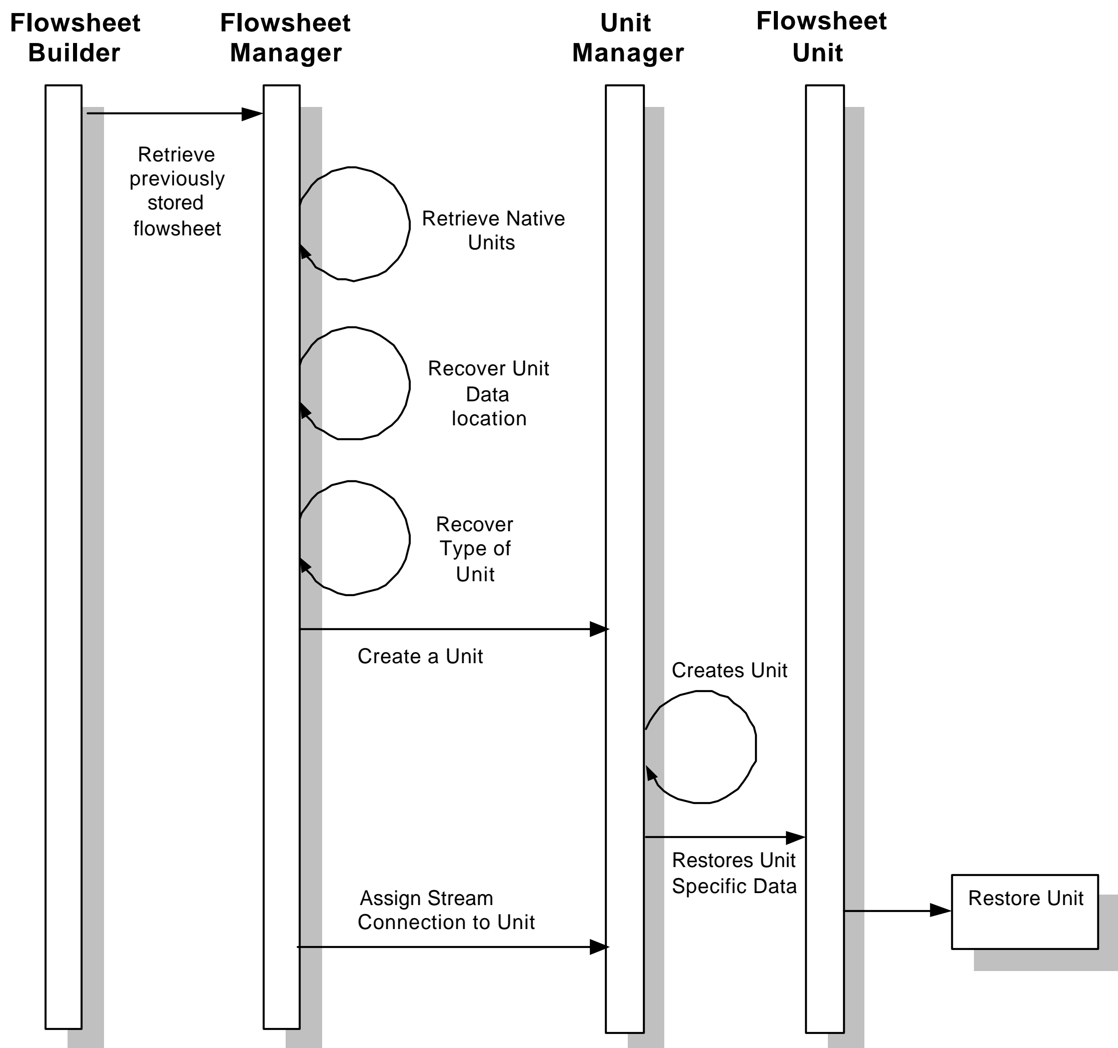


Figure 2.18 SQ – Retrieve Flowsheet

2.3.6 Save Flowsheet (ref. SQ-31-006)

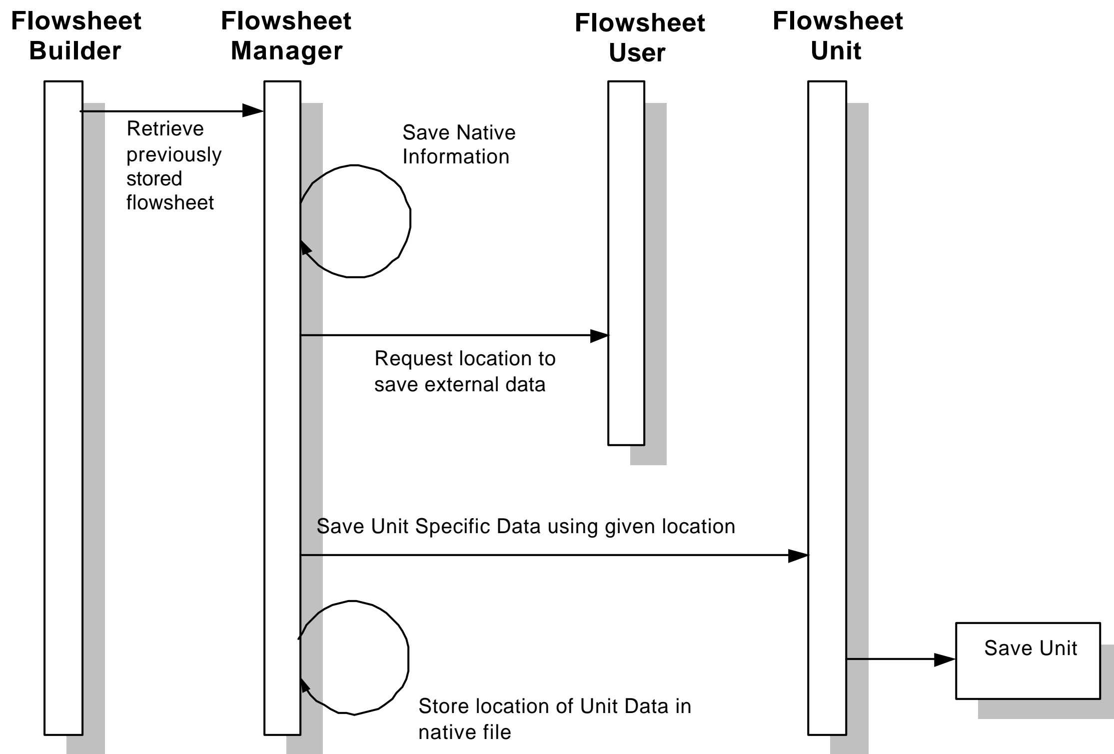


Figure 2.19 SQ – Save Flowsheet

2.3.7 Set Unit Specific Data (ref. SQ-31-007)

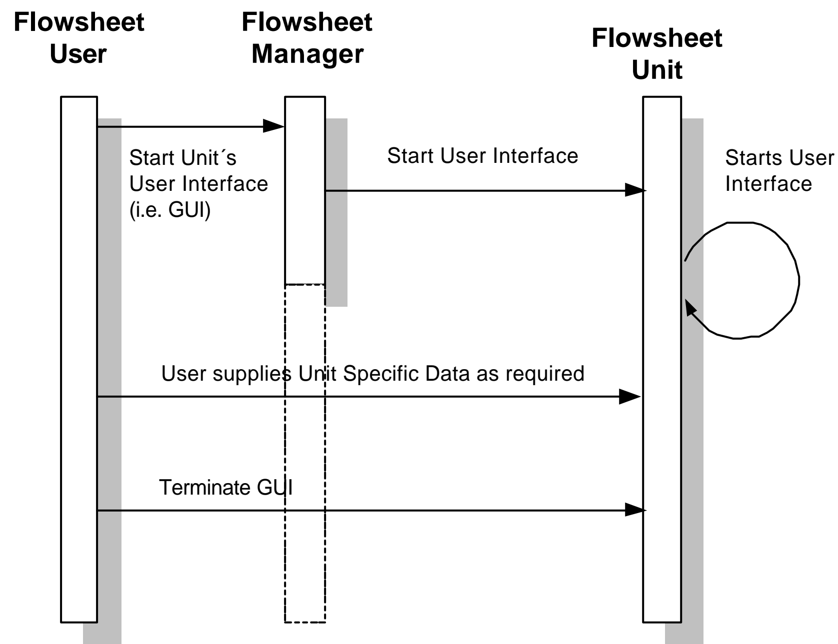


Figure 2.20 SQ – Set Unit Specific Data; A: Graphic User Interface available

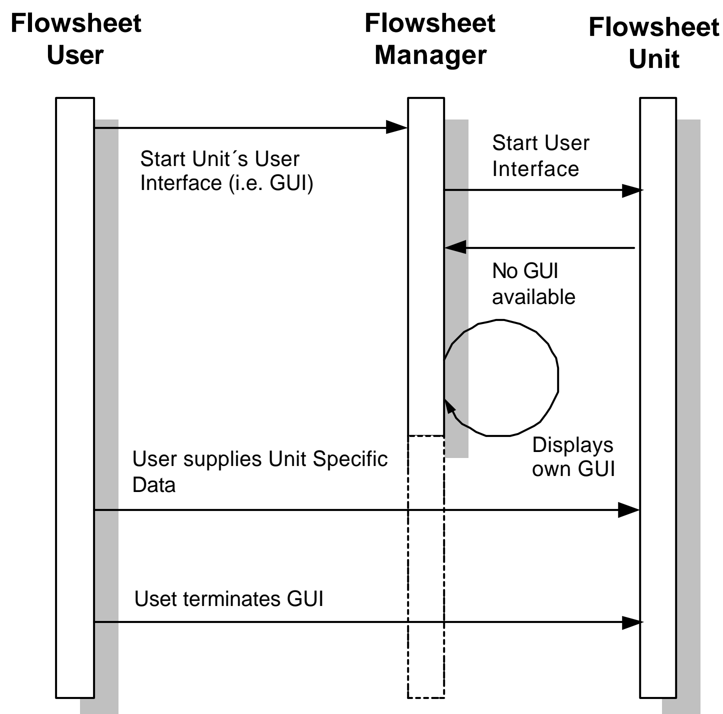


Figure 2.21 Set Unit Specific Data; B: Graphic User Interface not available

2.3.8 Specify Unit's Material Connections (ref. SQ-31-008)

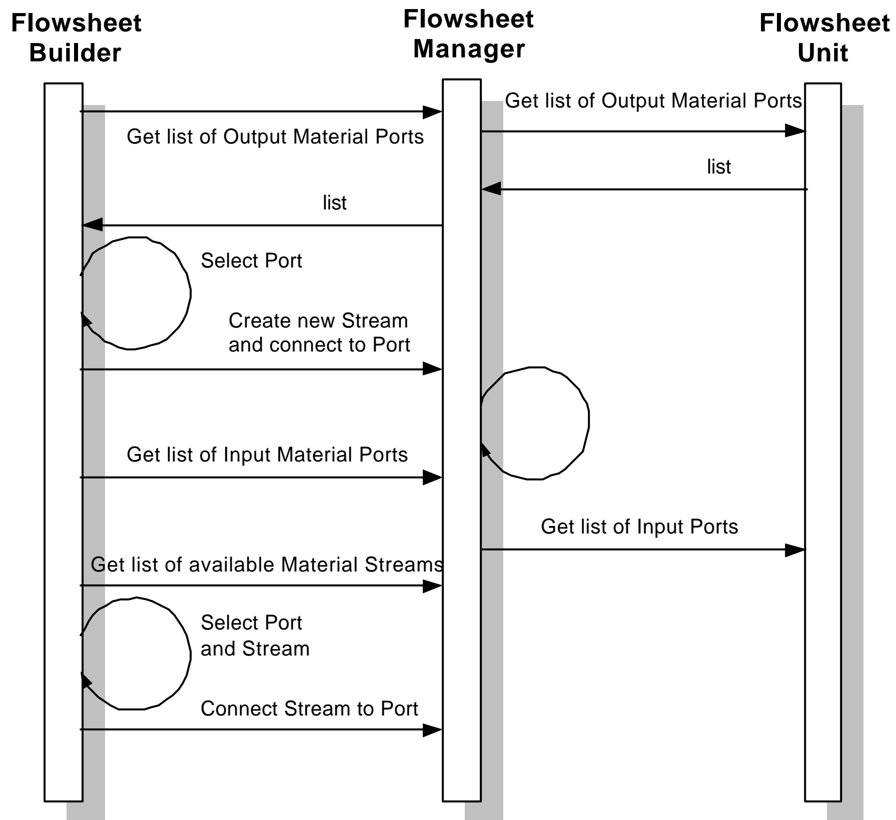


Figure 2.22 SQ – Specify Units Material Connections

2.3.9 Specify Unit's Information Connections (ref. SQ-31-009)

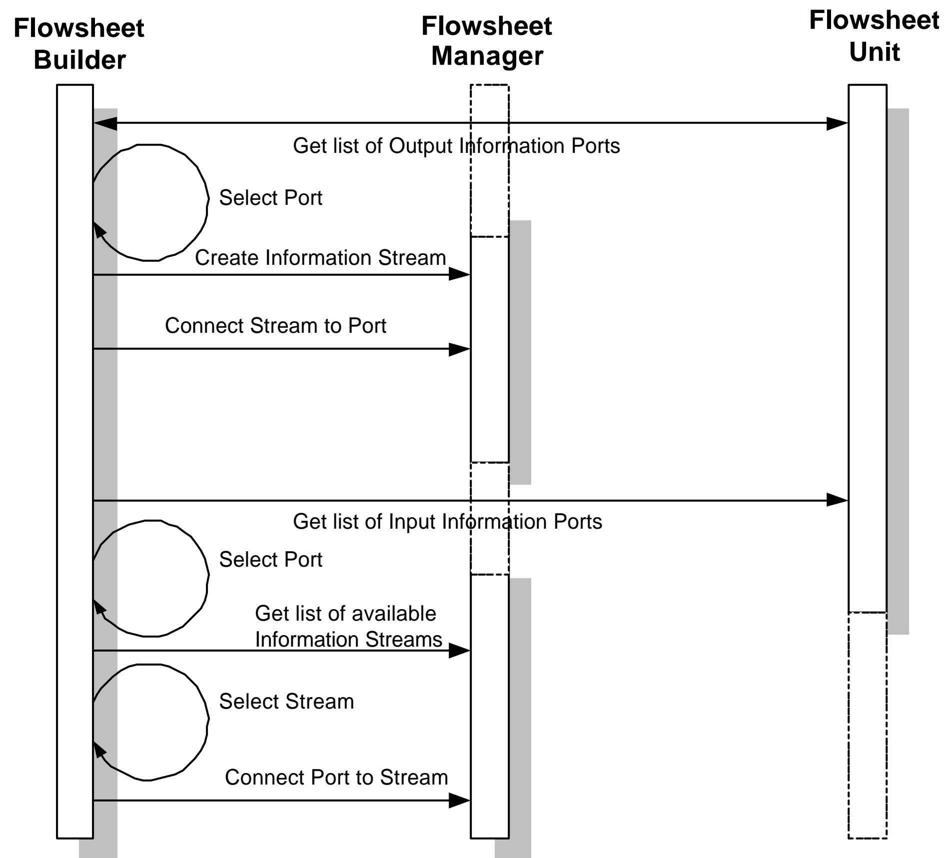


Figure 2.23 Set Unit Information Connections

3 Analysis

3.1 OVERVIEW
3.2 INTERFACE DIAGRAMS
3.3 SEQUENCE DIAGRAMS
3.4 STATE DIAGRAMS
3.5 COMPONENT DIAGRAM
3.6 INTERFACES DESCRIPTIONS
3.7 SCENARIOS

3.1 Overview

This chapter introduces the analysis models developed by the project team for the Unit Operations Group 1 sub-task. These models are described by a combination of text and UML diagrams, to show the solutions derived for the requirements expressed in the Use Cases. The UML diagrams presented are the interface, state and component diagrams, as well as some explanatory sequence diagrams.

3.2 Interface diagrams

This section presents the interface diagrams for UNIT Group 1.

3.2.1 UNIT Group 1 Interface Diagram (ref. IN-31-001)

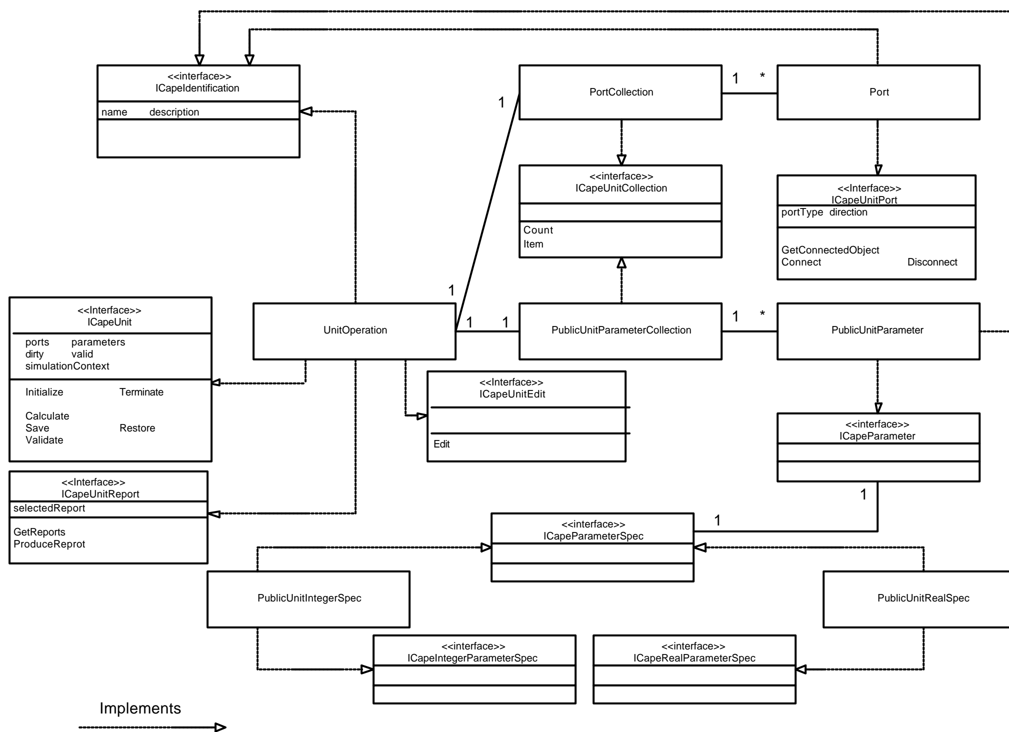


Figure 3.1 UNIT Group 1 Interface Diagram

3.3 Sequence diagrams

As before, only exceptions that result in non-trivial behaviour are shown and no returning responses from messages sent from one object to another are included.

3.3.1 Connecting a Port (ref. SQ-31-010)

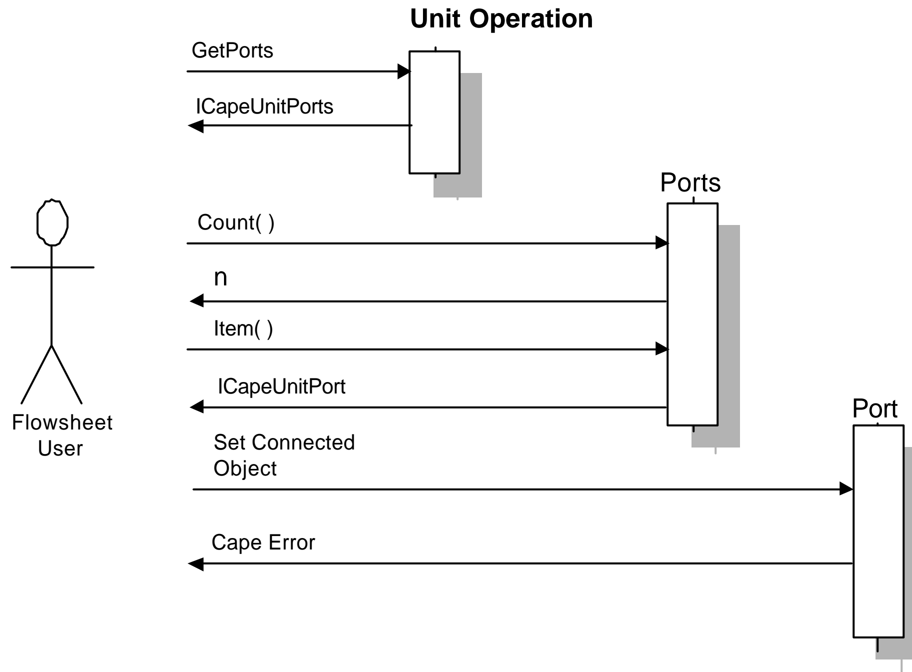


Figure 3.2 SQ – Connecting a Port

3.3.2 Edit/Viewing Unit's Specific Data (ref. SQ-31-011)

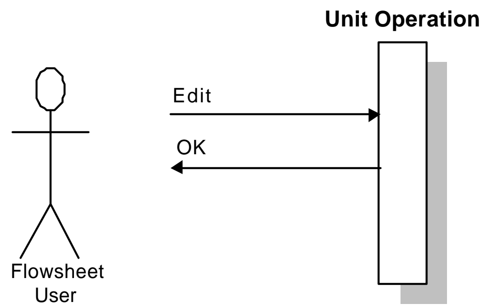


Figure 3.3 SQ – Edit/Viewing Unit Specific Data; (a) UO provides its own GUI for its Parameters

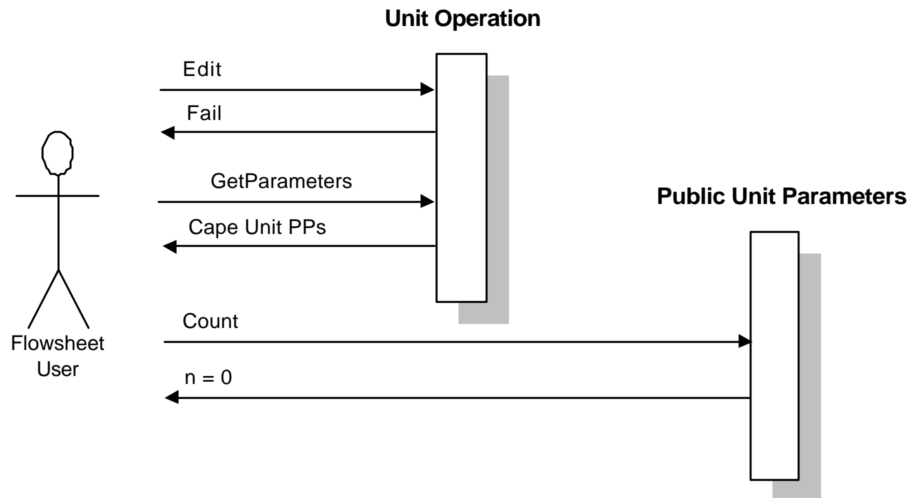


Figure 3.4 SQ – Edit/Viewing Unit Specific Data; (b) Unit does not provide a GUI and has no Parameters

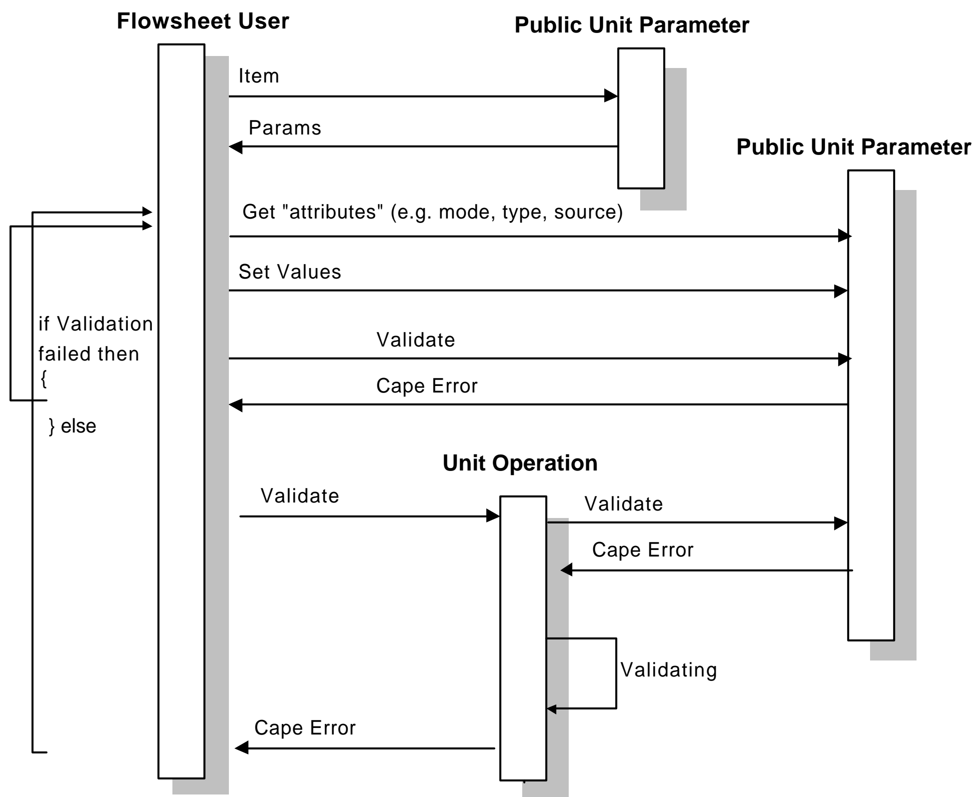


Figure 3.5 SQ – Edit/Viewing Unit Specific Data; (c) Unit does not provide a GUI, but does possess Public Unit Parameters. Host creates own GUI to allow user to enter values. Host sets Public Unit Parameters when GUI is closed

3.3.3 Calculate (ref. SQ-31-012)

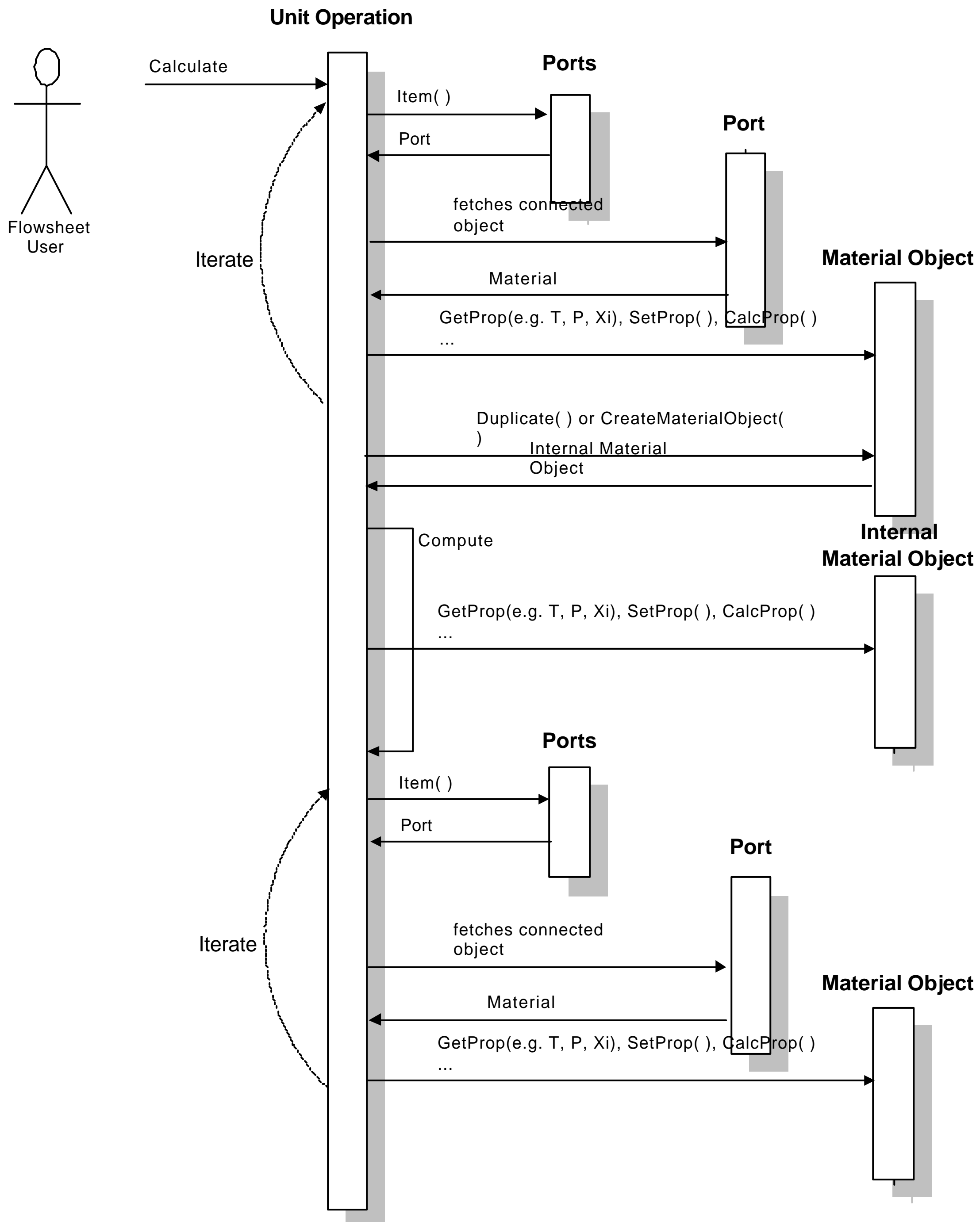


Figure 3.6 SQ – Calculating

3.4 State diagrams

This section shows the 4 main components used in UNIT: Port, Unit Operation, Parameter and Collection as shown in the interface diagram in section 3.2.1(ref. IN-31-001).

States indicated by a double rounded square represent: a) intermediate states of special importance that occur during the transition between two main states (the client does not necessarily need to be aware of the existence of the intermediate state), or b) states that are the result of an action performed upon a non-CO interface (e.g. performed on a private interface between Ports and/or Collections and/or Unit Operations).

Notice that, for instance, a user request for accessing the object (i.e. a Material Object) connected to a specific Port result in the state of the Port changing from "Connected" to "Delivering". This change of state is a direct consequence of a client action performed upon one of the methods of the interface ICapeUnitPort (i.e. the method GetConnectedObject) and it is a transition between two main states that are visible to the client.

Conversely, if a client requests that a Port be connected to an object, the state of the port will change from "Disconnected" to "Connected", as expected. However, this will occur only if the object sent to the Port is of the appropriate type (e.g. a Material Port is expects to be connected to Material Objects and nothing else). This very likely implies the existence of a state, here named as "Checking", in which the Port object will ensure that the connection can be performed. "Checking" is an intermediate state. In addition, a Port may need to notify its parent collection of the connection request. This is reflected in the intermediate state "Notifying Connection"

Finally, the transition of a Port from its creation to its "Disconnected" state takes place through the states "Non-Initialised" and "Initialised". This transition is provoked internally by the owner Unit Operation and/or the parent collection, and therefore is not the result of an external client action.

3.4.1 Port State Diagram (ref. ST-31-001 20)

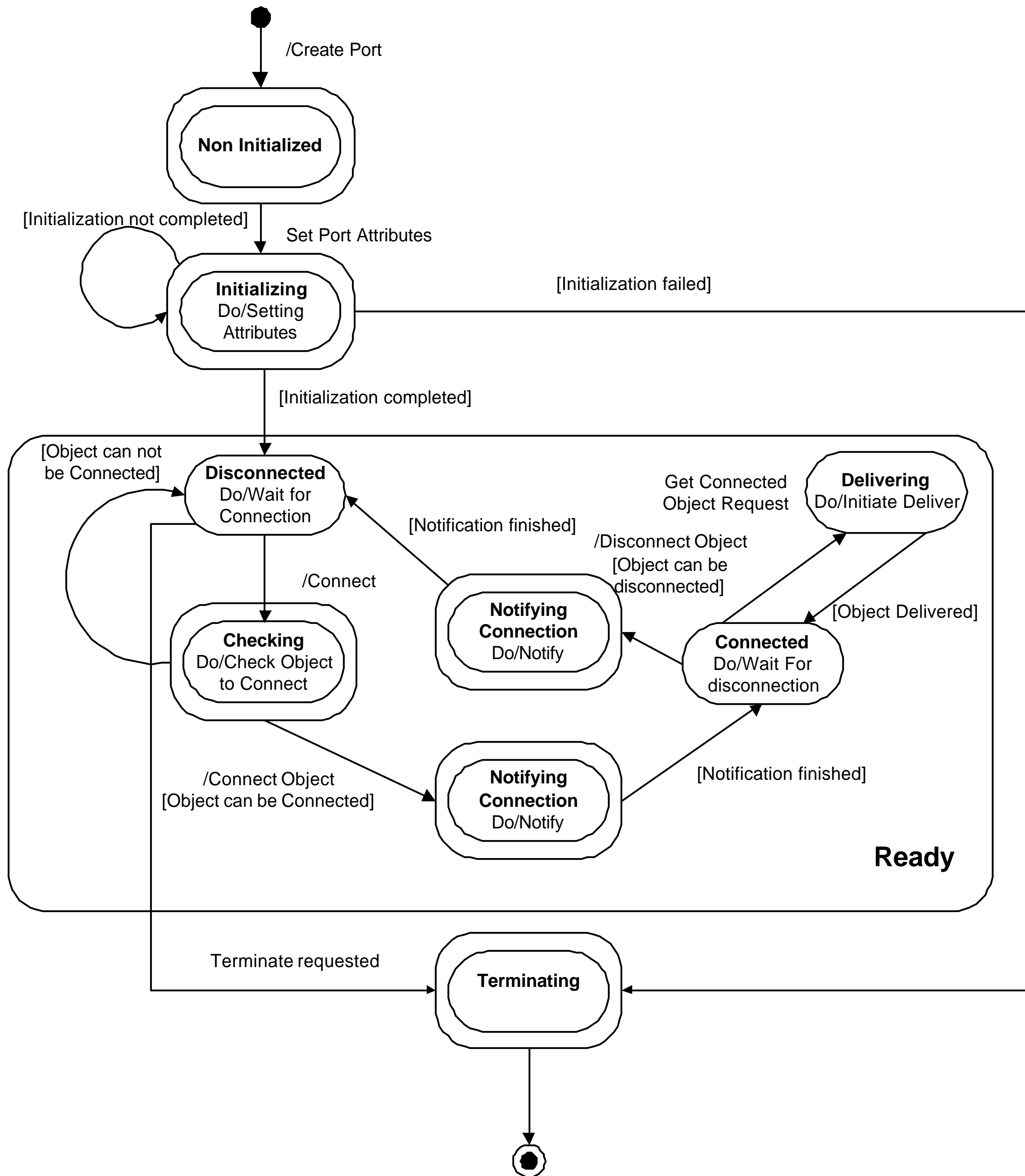


Figure 3.7 Port State Diagram

3.4.2 Collection of ports and parameters State Diagrams (ref. ST-31-001)

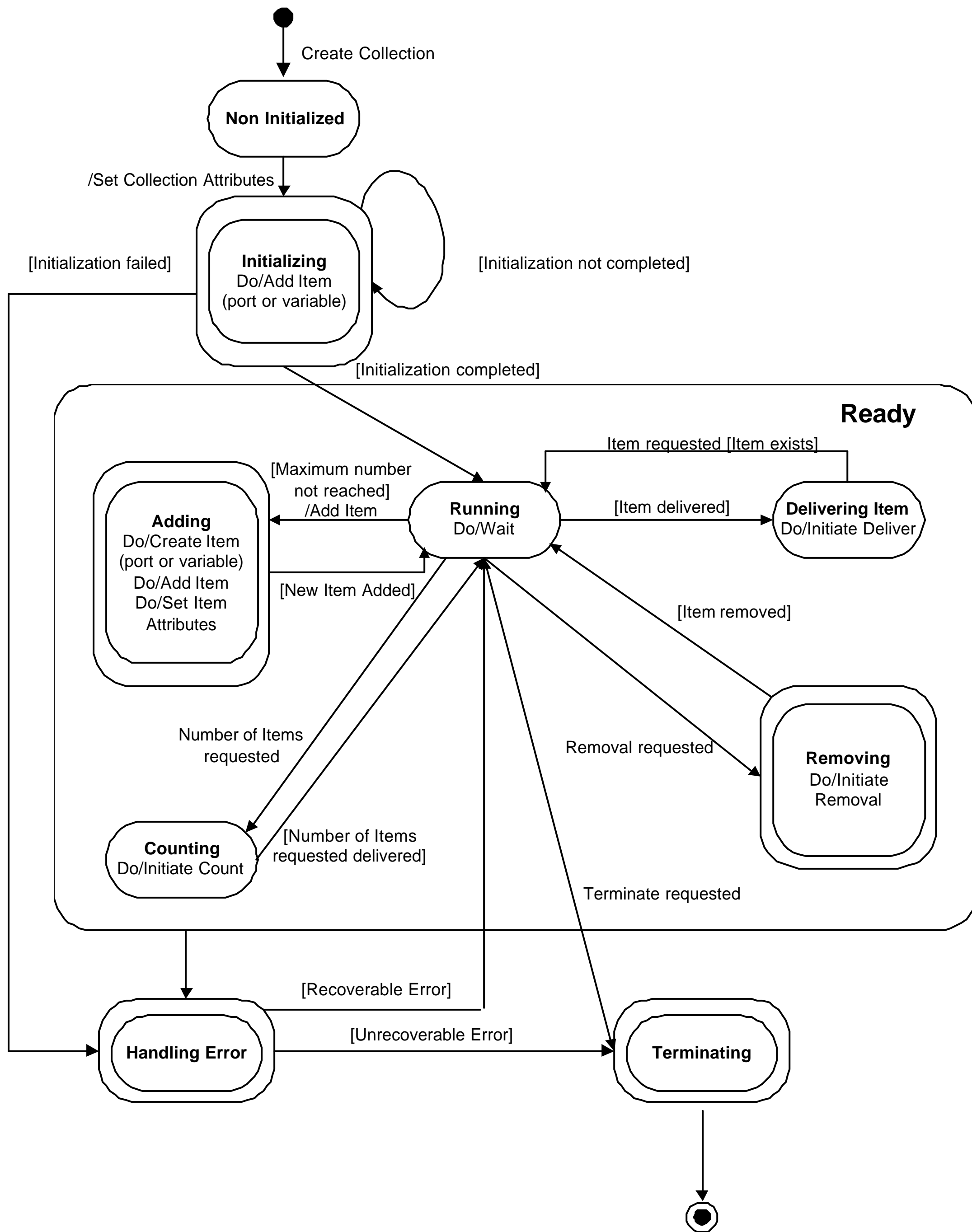


Figure 3.8 Collection of Parameters State Diagram

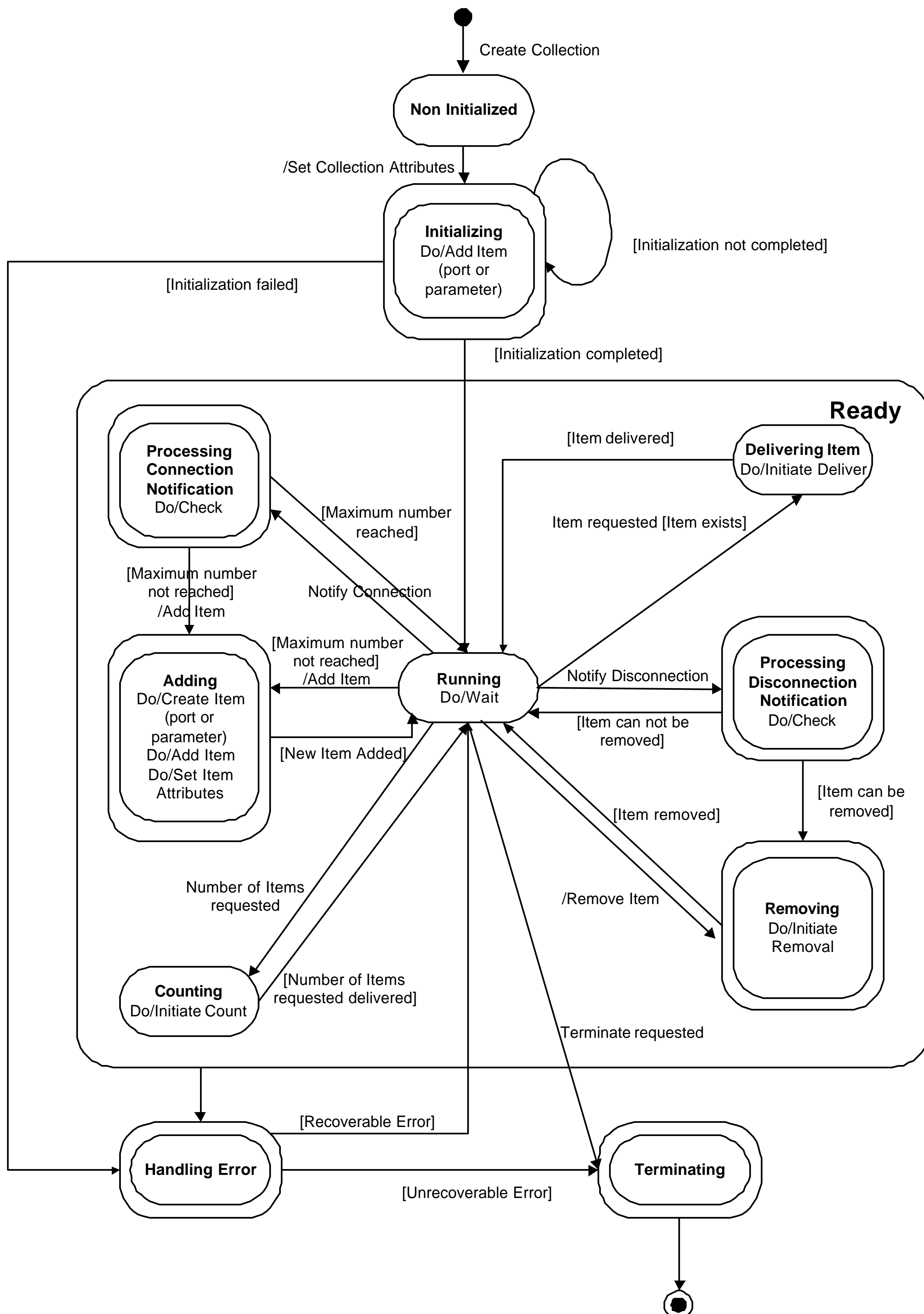


Figure 3.9 Collection of Ports State Diagram

3.4.3 Unit Operation State Diagram (ref. ST-31-001 20)

The following diagram shows activities and states in both the Simulator Executive and the Flowsheet Unit, which are running concurrently. This helps to clarify the context within which the Flowsheet Unit is operating. Only the relevant states in the simulator are shown.

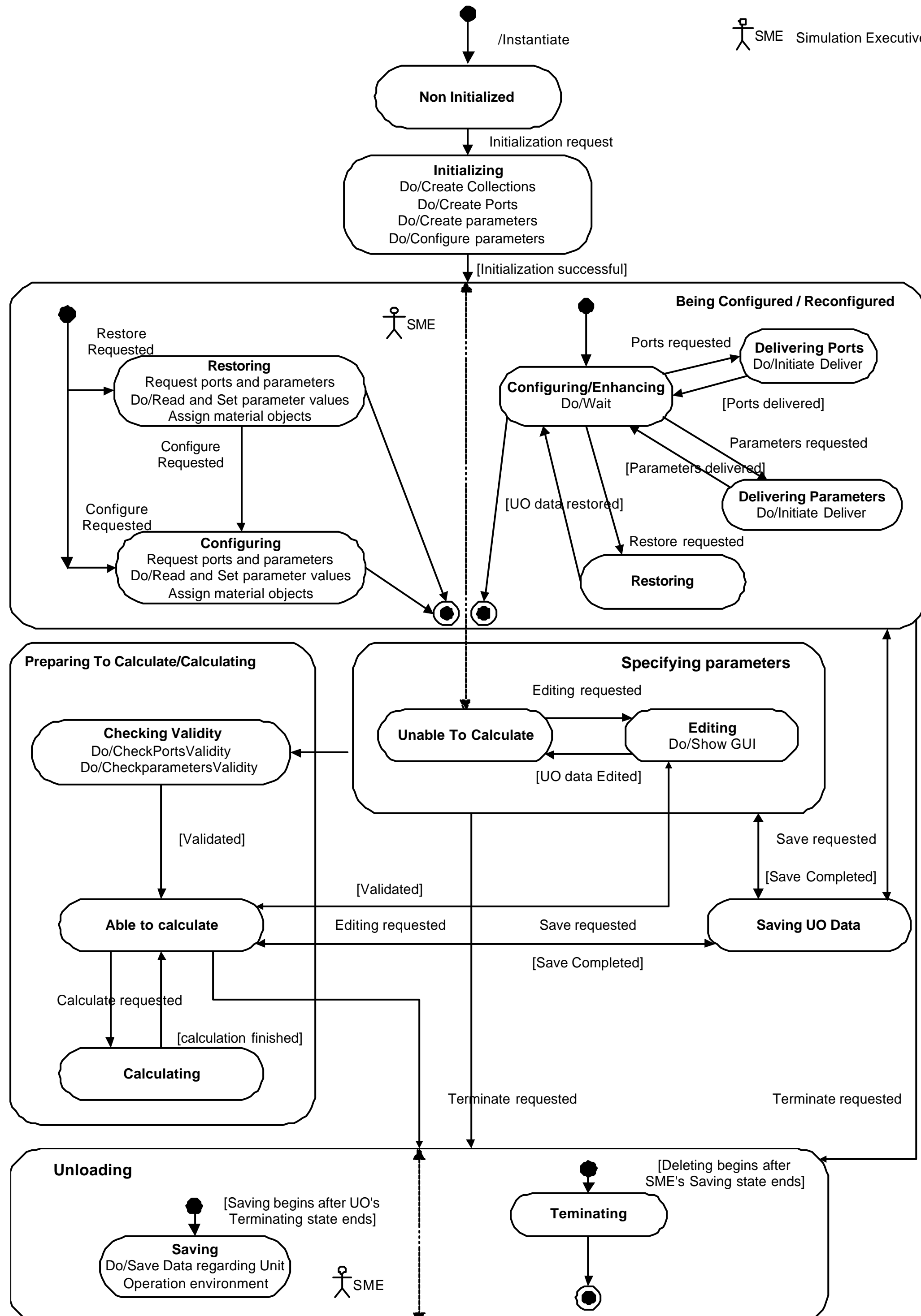


Figure 3.10 Unit Operation State Diagram

3.5 Component diagram

While the set of interfaces that are described below represents the functionality that all potential clients of CAPE OPEN interfaces for Unit Operations will be able to use, component diagrams identify how that functionality can be practically obtained. This is achieved by assigning responsibilities to real entities, rather than abstract ones (interfaces).

Thus, while the different interfaces and the factoring that was used, represent a way of logically dividing the required functionality in different abstract entities, the component diagram (and object model) below represent what is seen as a natural way of locating interfaces in living entities.

It is reasonable that different groups of people may decide to utilise and apply the CAPE-OPEN interfaces for Unit Operations, in a way that is finally reflected in slightly different object models. The above can be seen as a valid statement, as long as the client remains unaware of this and the functionality of interfaces is entirely provided.

In fact, the interfaces were designed to follow a tree structure that allows navigation from the top-most interface (ICapeUnit) to the bottom-most interface (ICapeParameterSpec). Every interface is provided with a method that allows the client to access an interface just at the lower following level. Thus, GetPorts and GetParameters return an IDispatch pointer that can be used to query the corresponding ICapeUnitCollection interface (i.e. QI(ICapeUnitCollection) will succeed), or alternatively to query the corresponding ICapeIdentification interface that provides information about the identity of the entity that represents the collection (i.e. its name and its description).

Note - for abbreviation we use the colloquial notation QI to mean QueryInterface, which is one of the member functions of IUnknown, and therefore of every COM object.

Similarly, the Item method in ICapeUnitCollection can be used to obtain an IDispatch pointer from which QI(ICapeUnitPort), if the collection is a collection of ports, or QI(ICapeParameter), if the collection is a collection of parameters, will succeed. As in the former case, alternatively the client may want to query ICapeIdentification, rather than ICapeUnitPort, by using the IDispatch pointer obtained through the Item method. This will give information about the identity of a specific port, or a specific parameter, and therefore QI(ICapeIdentification) has also to succeed here.

What follows, as a corollary, is that every registered CAPE-OPEN standard component, used to deliver the CAPE-OPEN interfaces has to support the ICapeIdentification interface. This is a necessary condition but not a sufficient one, since other non-standardised components used to deliver the CAPE-OPEN interfaces may also implement this interface. This is the case for a component such as Ports. A CAPE-OPEN component has other characteristics, such as that they are registered using CAPE-OPEN mechanisms (e.g. specific registry sub-keys or registered as belonging to certain CAPE-OPEN categories) that make them CAPE-OPEN plug-and-play components.

The above argument shows that there is a behaviour associated with the method QI, which is expected and which must be provided if the component is to be considered a well behaved CAPE-OPEN component. It is not enough to implement QI (which, by the way, is compulsory in the world of COM): QI must be implemented to provide a particular behaviour. For instance,

QI(ICapeIdentification) is required to return a valid pointer, if the component is a CAPE-OPEN one.

This tree structure for navigating from one interface to the others is part of the CAPE OPEN standard (and it is reflected in the sequence diagrams above), and therefore should be respected by the underlying object models and implementations. The component model depicted below is merely an example that takes into account this calling pattern and does not imply that the standard requires these additional components (other than UnitOperation) to be present.

The diagram below shows an overview of the various components, in the CAPE-OPEN Interface System, that were designed to implement the mixer-splitter prototype. It also shows the associations between the components, e.g. the UnitOperation component makes use of the PortCollection component, which makes use of the Port component.

The components defined by this specification are:

- **UnitOperation**: the unit operation itself.
- **Port**: the means by which a Flowsheet Unit is connected to its streams. Streams are implemented by means of material objects.
- **PortCollection**: the means by which the Flowsheet Unit groups together its ports.
- **Parameter**: a Public Unit Parameter is the means by which a component can make its internal variables visible to another component.
- **Parameter Collection**: the collection of Public Unit Parameters that a Flowsheet Unit wishes to expose to the outside world.

The components defined by THRM are:

- **Property Package System**: the thermodynamic package, which may be built into the simulator, or be a CAPE-OPEN plug-in component itself.
- **Equilibrium Server**: the component that performs flash calculations.
- **Calculation Routine**: the means to implement custom thermodynamic calculations
- **Material Object**: represents a flowsheet stream and provides access to all thermodynamic calculations. It is provided by the Simulator Executive.
- **Material Template Object**: provides the base functionality of a material object. Templates contain the component list, property package etc., for the class of streams they represent.

The final component, which is shown on the diagram, is not actually required for the mixer-splitter implementation, but is included for completeness:

- **Simulation Executive Environment**: the host simulator.

The design of the unit interface uses a number of separate components, rather than a single unit component because:

- It was felt to be conceptually clearer to have one component for one concept.
- Having a separate parameters component would make it easier to reuse it in other parts of the system, if required.

However, this area may be subject to revision, as the prototypes are evaluated.

Note that the port and parameters collections are represented as two separate components, whereas, in fact, they both use a generic collection component. Also, the diagram is presented in terms of COM. In particular, components are shown using the IDispatch interface of the components with which they interact. IDispatch along with IUnknown provides access to all the interfaces on a component, so the diagram can be shown in a less cluttered form.

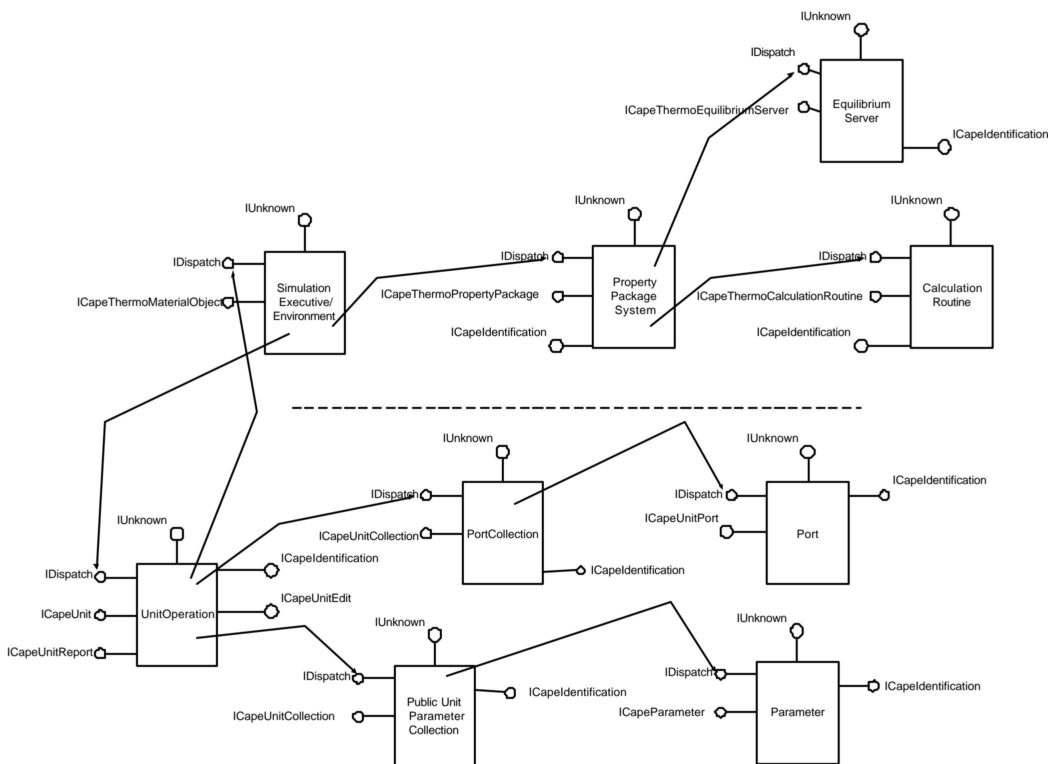


Figure 3.11 Unit Operations Components Diagram, showing the interaction with other CO Components

The components below the dotted line are described in this specification. Those above (except for the simulator) are described in the THRM specification.

3.6 Interface descriptions

3.6.1 Interface ICapeUnit
3.6.2 Interface ICapeUnitEdit
3.6.3 Interface ICapeUnitPort
3.6.4 Interface ICapeUnitCollection
3.6.5 Interface ICapeUnitReportnt
3.6.6 Interface ICapeIdentification
3.6.7 Interface ICapeUnitPortVariables

Each interface is presented together with its corresponding methods.

The UNIT Group1 interfaces are:

- **ICapeUnit**. This interface handles most of the interaction with the Flowsheet Unit.
- **ICapeUnitEdit**. This interface allows the COSE to display any graphical user interfaces the Flowsheet Unit may have.
- **ICapeUnitPort**. This interface provides access to a Flowsheet Units port. These in turn provide access to materials, energy and information streams provided by the host simulator.
- **ICapeUnitCollection**. This interface provides a means of collecting together lists of CAPE-OPEN entities such as parameters and ports.
- **ICapeUnitReport**. This interface provides access to the reporting facilities of the Flowsheet Unit. This interface may be subject to change later on in the project as reporting becomes more fully defined and able to handle richer forms of reporting such as graphics.
- **ICapeIdentification**. This interface provides a means of obtaining the name and description of a CAPE-OPEN component.

Summary of changes at the 0-9-3 revision of the specification.

- The GetDirty and GetValid methods have been merged into a single method GetValStatus. This method returns a value of type CapeValidationStatus, which is an enumeration of three values – CAPE_VALID, CAPE_INVALID and CAPE_NOT_VALIDATED. See the description of the GetValStatus method for an explanation of the meaning of these values. This change makes the validation for units and parameters consistent.

- A number of methods, Calculate, Restore, Save, Initialize and Terminate, all returned a string value and a boolean value (as well as an HRESULT in the COM implementation). The string value was intended to return a message in the event of a failure. These arguments have been removed with the introduction of a standard error handling strategy which is documented in the "Error Common Interfaces" specification document produced by the Methods and Tools Group.

- Unit operations are expected to implement the "Error Common Interfaces" specification, so the description of each method of each interface now gives a list of the errors that the method may raise.

- Unit operations are expected to return parameter collections containing parameter objects that implement the 0-9-3 Parameter specification. These parameter interfaces are documented in the "Parameter Common Interfaces" specification document produced by the Methods and Tools Group.

3.6.1 Interface ICapeUnit Methods

- **GetParameters**

Interface Name	ICapeUnit
Method Name	GetParameters
Returns	CapeError

Description

Return the collection of Public Unit Parameters (i.e. ICapeUnitCollection).

These are delivered as a collection of elements exposing the interface ICapeParameter. From there, the client could extract the ICapeParameterSpec interface or any of the typed interfaces such as ICapeRealParameterSpec, once the client establishes that the Parameter is of type double.

Arguments

Name	Type	Description
[out, return] publicParams	CapeInterface	The interface of the collection containing the list of Public Unit Parameters.

Errors:

ECapeUnknown,

ECapeFailedInitialisation,

ECapeBadInvOrder,

ECapeBadCOParameter

GetValStatus

Interface Name	ICapeUnit
Method Name	GetValStatus
Returns	CapeValidationStatus

Description

Get the flag that indicates whether the Flowsheet Unit is valid (e.g. some parameter values have changed but they have not been validated by using Validate). It has three possible values:

- notValidated(CAPE_NOT_VALIDATED): the unit's validate() method has not been called since the last operation that could have changed the validation status of the unit, for example an update to a parameter value of a connection to a port.
- invalid(CAPE_INVALID): the last time the unit's validate() method was called it returned false.
- valid(CAPE_VALID): the last time the unit's validate() method was called it returned true.

Arguments

Name	Type	Description
[out, return] ValStatus	CapeValidationStatus	CAPE_VALID meaning the Validate method returned success CAPE_INVALID meaing the Validate method returned failure CAPE_NOT_VALIDATED meaning that the Validate method needs to be called to determine whether the unit is valid or not.

Errors:

ECapeUnknown,

ECapeInvalidArgument

Initialize

Interface Name	ICapeUnit
Method Name	Initialize
Returns	CapeError

Description

The Flowsheet Unit initialises itself. Any initialisation that could fail must be placed here. Initialize is guaranteed to be the first method called by the client. Initialize has to be called once when the Flowsheet Unit is instantiated in a particular flowsheet.

Note that the Initialize method is intended to implement software initialization not to perform an initial solution of the Flowsheet Unit. It is expected that the method would be used to create and configure ports and parameters and to define default values for variables and parameters.

There are no input or output arguments for this method.

Errors:

ECapeUnknown,

ECapeLicenceError,

ECapeFailedInitialisation,

ECapeOutOfResources

❑ Calculate

Interface Name	**IcapeUnit**
Method Name	Calculate
Returns	CapeError

Description

The Flowsheet Unit performs its calculation, that is, computes the variables that are missing at this stage in the complete description of the input and output streams and computes any public parameter value that needs to be displayed. Calculate will be able to do progress monitoring and checks for interrupts as required using the simulation context. At present, there are no standards agreed for this.

It is recommended that Flowsheet Units perform a suitable flash calculation on all output streams. In some cases a Simulation Executive will be able to perform a flash calculation but the writer of a Flowsheet Unit is in the best position to decide the correct flash to use.

Before performing the calculation, this method should perform any final validation tests that are required. For example, at this point the validity of Material Objects connected to ports can be checked

There are no input or output arguments for this method.

Errors:

ECapeUnknown,

ECapeBadInvOrder,

ECapeOutOfResources,

ECapeTimeOut,

ECapeSolvingError,

ECapeLicenceError

❑ GetPorts

Interface Name	IcapeUnit
Method Name	GetPorts
Returns	CapeError

Description

Return an interface to a collection containing the list of unit ports (e.g. ICapeUnitCollection).

Return the collection of unit ports (i.e. ICapeUnitCollection). These are delivered as a collection of elements exposing the interfaces ICapeUnitPort

Arguments

Name	Type	Description
[out, return] portsInterface	CapeInterface	A reference to the interface on the collection containing the specified ports

Errors:

ECapeUnknown,

ECapeFailedInitialisation,

ECapeBadInvOrder

❑ Terminate

Interface Name	ICapeUnit
Method Name	Terminate
Returns	CapeError

Description

The Flowsheet Unit releases all of its allocated resources. This is called before the object destructor. Terminate may check if the data has been saved and return an error if not.

There are no input or output arguments for this method.

Errors:

ECapeUnknown,

ECapeBadInvOrder

Restore

Interface Name	**ICapeUnit**
Method Name	Restore
Returns	CapeError

Description

The Flowsheet Unit is restored from a previously saved state. The [in] argument identifies the location from which the Flowsheet Unit should read its data. Here, data refers to whatever data the writer of the Flowsheet Unit chooses to save. This location may bear no relation to any location to which data has previously been saved. It can be a CapeString type, identifying a full path of an ASCII file, or it can be a type CapeVariant, hosting a reference to any standard COM interface for persistence such as IStorage or IStream.

It is strongly recommended that the IStorage, or IStream variations are not used in a COM environment because a Flowsheet Unit written in Visual Basic (versions 6 and lower) is not able to make use of an IStorage or IStream pointer. In a COM environment the preferred solution is for the Flowsheet Unit and the COSE to support the standard COM persistence interfaces and protocols. See section 5.2.8 for a discussion of this recommendation.

Arguments

Name	Type	Description
[in] storage	CapeVariant	A reference to the place where the Flowsheet Unit data has been stored.

Errors:

ECapeUnknown,

ECapeInvalidArgument,

ECapePersistenceNotFound,

ECapeIllegalAccess,

ECapeNoImpl

❑ Save

Interface Name	ICapeUnit
Method Name	Save
Returns	CapeError

Description

The Flowsheet Unit saves its private data in the location indicated by the simulator (i.e. storage), that can be a given position in a structured document or a file name (CapeString type). The Flowsheet Unit is free to use the storage mechanism that the simulator provides, or to use its own internal mechanisms. A valid scenario would be that the Flowsheet Unit checks the type of the storage parameter passed in, accepting it if it is of type CapeString (e.g. an ASCII file name) or rejecting it if it is of another type that the Flowsheet Unit does not handle. In the latter case, the Flowsheet Unit may decide to create its own text file, save its data there, and send back the full path of the file to the simulator (notice the [in, out] argument) which stores that string along with its own simulation data.

It is recommended that in a COM implementation a component should support one of the standard COM persistence methods in addition to this method. See section 5.2.8 for a discussion of this recommendation.

A Flowsheet Unit must save its state completely so that the Restore can recreate that state. Flowsheet Unit authors should not rely on Validate being called after Restore. The same requirement applies when COM persistence methods are implemented.

Arguments

Name	Type	Description
[in, out] storage	CapeVariant	Reference to the storage where the Flowsheet Unit's data is saved e.g. File name or stream.

Errors:

ECapeUnknown,

ECapeInvalidArgument,

ECapePersistenceNotFound,

ECapeIllegalAccess,

EcapeNoImpl

❑ Validate

Interface Name	ICapeUnit
Method Name	Validate
Returns	CapeError

Description

Sets the flag that indicates whether the Flowsheet Unit is valid by validating the ports and parameters of the Flowsheet Unit. For example, this method could check that all mandatory ports have connections and that the values of all parameters are within bounds.

Note that the Simulation Executive can call the Validate routine at any time, in particular it may be called before the executive is ready to call the Calculate method. This means that Material Objects connected to unit ports may not be correctly configured when Validate is called. The recommended approach is for this method to validate parameters and ports but not Material Object configuration. A second level of validation to check Material Objects can be implemented as part of Calculate, when it is reasonable to expect that the Material Objects connected to ports will be correctly configured.

Arguments

Name	Type	Description
[out, return] isValid	CapeBoolean	TRUE if the Unit is valid
[out] message	CapeString	An optional message describing the cause of the validation failure.

Errors:

ECapeUnknown,

ECapeBadCOParameter,

ECapeBadInvOrder

SetSimulationContext

Interface Name	ICapeUnit
Method Name	SetSimulationContext
Returns	CapeError

Description

The simulation context is assigned to the Flowsheet Unit. (This is to be defined more precisely later)

Arguments

Name	Type	Description
[in] simulationContext	CapeInterface	the context of the simulator (progress, thermo, numerics)

Errors:

ECapeUnknown

3.6.2 Interface ICapeUnitEdit Methods

- **Edit**

Interface Name	**ICapeUnitEdit**
Method Name	Edit
Returns	CapeError

Description

The Flowsheet Unit displays its user interface and allows the Flowsheet User to interact with it. If no user interface is available it returns an error.

There are no input or output arguments for this method.

Errors:

ECapeUnknown

ECapeNoImpl

ECapeBadInvOrder

3.6.3 Interface ICapeUnitPort Methods

- **GetPortType**

Interface Name	**ICapeUnitPort**
Method Name	GetPortType
Returns	CapeError

Description

Returns the type of the port. The returned value has to be one of the constants defined as of the type CapePortType (i.e.. CapeMaterial, CapeEnergy, CapeInformation or CapeAny). PortType allows the client to know what is the type of Objects the Port is expecting.

Arguments

Name	**Type**	**Description**
[out, return] portType	CapePortType	The type of the port (i.e. CapeMaterial, CapeEnergy, CapeInformation or CapeAny)

Errors:

ECapeUnknown

ECapeFailedInitialisation

❑ GetDirection

Interface Name	**ICapeUnitPort**
Method Name	GetDirection
Returns	CapeError

Description

Returns the direction of the Port. This attribute of a Port indicates the direction in which objects or information connected to the port are expected to flow (e.g. Material, Energy or Information Objects). The returned value has to be one of the constants defined as of the type CapePortDirection (e.g. CapeInput, CapeOutput or CapeInputOutput).

Arguments

Name	**Type**	**Description**
[out, return] portDirection	CapePortDirection	The direction of the port (e.g. CapeInput, CapeOutput or CapeInputOutput).

Errors:

ECapeUnknown

ECapeFailedInitialisation

GetConnectedObject

Interface Name	**ICapeUnitPort**
Method Name	GetConnectedObject
Returns	CapeError

Description

Returns the object that is connected to the Port. A client is provided with the Material, Energy or Information object that was previously connected to the Port, using the Connect method.

Arguments

Name	Type	Description
[out, return] connectedObject	CapeInterface	The object that is connected to the port. It can be one among Material, Information or Energy Objects

Errors:

ECapeUnknown

ECapeFailedInitialisation

❑ Connect

Interface Name	IcapeUnitPort
Method Name	Connect
Returns	CapeError

Description

Method used by clients, when they request that a Port connect itself with the object that is passed in as argument of the method. Probably, before accepting the connection, a Port will check that the Object sent as argument is of the expected type and according to the value of its attribute portType.

Arguments

Name	Type	Description
[in] objectToConnect	CapeInterface	The object that is wanted to be connected to the port (i.e. Material, Energy or Information Objects).

Errors:

ECapeUnknown

ECapeInvalidArgument

❑ Disconnect

Interface Name	**IcapeUnitPort**
Method Name	Disconnect
Returns	CapeError

Description

Disconnects the port from whichever object is connected to it.

There are no input or output arguments for this method.

Errors:

ECapeUnknown

3.6.4 Interface ICapeUnitCollection Methods

- **Item**

Interface Name	**ICapeUnitCollection**
Method Name	Item
Returns	CapeError

Description

Return an element from the Ports collection or the collection of unit parameters. The requested element can be identified by its actual name (e.g. type CapeString) or by its index in the collection (e.g. type CapeLong).

Arguments

Name	**Type**	**Description**
[in] id	CapeVariant	Identifier for the requested item: 1) name of item 2) position in collection
[out, return] item	CapeInterface	The requested element.

Errors:

ECapeUnknown

EcapeFailedInitialisation

EcapeInvalidArgument

ECapeOutOfBounds

❑ Count

Interface Name	ICapeUnitCollection
Method Name	Count
Returns	CapeError

Description

Return the number of items in the collection.

Arguments

Name	Type	Description
[out, return] itemsCount	CapeLong	Number of items in the collection

Errors:

ECapeUnknown

EcapeFailedInitialisation

3.6.5 Interface ICapeUnitReport Methods

- **Get/SetSelectedReport**

Interface Name	**ICapeUnitReport**
Method Name	Get/SetSelectedReport
Returns	CapeError

Description

Return/set the active report in the Flowsheet Unit.

Arguments

Name	**Type**	**Description**
[out, return]/[in] selectedReport	CapeString	The active report

Errors:

ECapeUnknown

ECapeNoImpl

- **GetReports**

Interface Name	**ICapeUnitReport**
Method Name	GetReports
Returns	CapeError

Description

Return the list of available Flowsheet Unit reports.

Arguments

Name	**Type**	**Description**
[out, return] reports	CapeArrayString	The list of possible reports.

Errors:

ECapeUnknown

ECapeNoImpl

❑ ProduceReport

Interface Name	**ICapeUnitReport**
Method Name	ProduceReport
Returns	CapeError

Description

Produce the designated report. If no value has been set, it produces the default report.

Arguments

Name	Type	Description
[in, out] report	CapeString	String containing the text for the currently selected report.

Errors:

ECapeUnknown

ECapeNoImpl

3.6.6 Interface ICapeIdentification Methods

The methods of the ICapeIdentification interface are now described in the "Identification CO Service " specification produced by the Methods and Tools Group.

3.6.7 Interface ICapeUnitPortVariables

This interface is optional and would be implemented by a port object. It is intended to allow a port to describe which Equation-oriented variables are associated with it and should only be implemented for the ports contained in a unit operation which supports the ICapeNumericESO interface described in "CAPE-OPEN Interface Specification – Numerical Solvers".

- **SetIndex**

Interface Name	**IcapeUnitPortVariables**
Method Name	SetIndex
Returns	CapeError

Description

SetIndex is used to set the indices of the variables occurring in a port. Typically this method is only used within a unit operation. It should not be used by any external software to change the indices of the port variables.

It is expected that a port will typically contain variables that represent a temperature, a pressure, an enthalpy, a flowrate, and a molar volume, with the composition represented by the molar fraction for each component. Other possibilities are allowed but it is the COSE's responsibility to determine whether it can make a connection to a port based on the variables that it contains.

Arguments

Name	**Type**	**Description**
[in] variable_type	CapeString	The type of the variable whose index is to be set. The value must be one of the recognised CAPE-OPEN Physical Property names.
[in] component	CapeString	For a fraction variable a component name must be supplied

[in] index	CapeLong	The index that identifies this variable in the ESO associated with the unit operation.

Errors:

ECapeUnknown

EcapeInvalidArgument

ECapeNoImpl

❑ GetIndex

Interface Name	IcapeUnitPortVariables
Method Name	GetIndex
Returns	CapeError

Description

GetIndex is used to get the index of a specific variable occurring in a port. Typically the client of a unit operation uses this method when it builds an equation-oriented representation of a simulation problem. The indices of the variables occuring in a port needs to be known in order to create the equations needed to represent connections to the unit.

It is expected that a port will typically contain variables that represent a temperature, a pressure, an enthalpy, a flowrate, and a molar volume, with the composition represented by the molar fraction for each component. Other possibilities are allowed but it is the COSE's responsibility to determine whether it can make a connection to a port based on the variables that it contains.

Arguments

Name	Type	Description
[in] variable_type	CapeString	The type of the variable whose index is to be set. The value must be one of the recognised CAPE-OPEN Physical Property names.
[in] component	CapeString	For a fraction variable a component name must be supplied
[out,return] index	CapeLong	The index that identifies this variable in the ESO associated with the unit operation.

Errors:

ECapeUnknown

EcapeInvalidArgument

ECapeNoImpl

3.7 Scenarios

This section describes the scenario of actions that was used to validate the set of interfaces for Unit Operations. This scenario exercises the most important methods of the UNIT interfaces, and so guarantees the core functionality expected from CAPE-OPEN Unit Operations interfaces, within the limitations of the Group 1 sub-task GRP1 (see Introduction). It is based on a mixer/splitter component, since this exhibits the interface behaviours of more complex Flowsheet Unit operations, but is simple to implement.

3.7.1 Mixer/Splitter Scenario (ref. SC-31-001)

The purpose of this scenario is to focus the scope of the interface specifications onto what is required to implement the mixer-splitter prototype. It provides the overall functionality required for this prototype, rather than an engineering specification of a mixer-splitter. Methods, which have been left out, will be dealt with in the GRP2 sub-task of the work package.

For the purposes of this exercise the following assumptions are made:

- The Flowsheet Unit can ask a material for the value of its enthalpy and get it. The host simulator may not actually store enthalpy with its streams, but it is assumed that the CAPE-OPEN interface supplies it by some means. In other words, we assume that we do not have to understand the normal internal arrangements of the simulator to obtain any CAPE-OPEN interface attributes.

- The unit can set the attributes of materials (e.g. T, P etc.) in “batch mode”, that is, an attribute can be set without triggering a recalculation of the material’s internal state. When the unit has finished setting up a material, it then asks it to recalculate. This is done to make the operation of the unit efficient by avoiding unnecessary calculations.

- The unit does not need to inquire about the availability of a thermodynamic server. It just asks the materials to perform actions and the material is assumed to invoke the thermodynamic server internally, if it needs to. In a production simulator, of course, the unit would need to do this. It is quite possible that there might be no physical property system connected at all. This is just a simplifying assumption for this initial prototype.

- All streams, both inlet, outlet and internal, have the same material template.

- The mixer-splitter allows the user to specify the split factors used to direct flow to the outlet streams. It also has the ability to specify heat input.

Add a unit to the flowsheet

- User selects a mixer-splitter from a palette of unit operations and places it on the flowsheet.

- Simulator asks the unit to initialise itself.

- The user selects or creates the streams and requests the simulator to connect them, one at a time, to the ports of the unit, possibly by using a GUI to drag the stream to a port on the unit icon or by interpretation of a text input file.

Enter Unit Specific Data

- The simulator does not know if a user interface (UI) is available from the unit, so that it must query the unit for one. If a UI is available, it is displayed. If not, the simulator attempts to construct one from the unit's list of Public Unit Parameters.
- When the input is complete the unit checks its parameters for validity and consistency.

Define unit report

- User asks the unit which reports it can produce.
- The unit gives a list from which the user selects.

Simulation Run

- Simulator asks the unit to calculate itself
- The unit retrieves the materials associated with its streams by requesting them from its ports.
- The unit creates an internal material object. This is to contain the mixed feeds. The unit creates an internal array to contain the mixed composition. This is done to minimise calls to the material object to set compositions.
- The unit requests the mixed stream and all the outlet streams to postpone calculation of their internal states until it has finished defining the compositions and conditions.
- The unit retrieves the component flows of each feed from the associated materials and adds them to the mixed stream composition array. When complete, the composition is assigned to the mixed stream material object.
- The unit fetches the enthalpy from each feed material, sums them and adds the specified heat input. The total enthalpy is assigned to the mixed stream.
- The unit sets the pressure of the mixed stream to the minimum of the feed pressures.
- The unit asks the mixed stream to calculate its temperature.
- The unit assigns molar and enthalpy flows to the output streams in accordance with the specified split factors.
- The unit assigns the composition of the mixed stream to the output streams.
- The unit assigns the pressure of the mixed stream to the output streams.
- The unit requests the output streams to calculate their temperatures.

Reporting

- Simulator asks the unit to produce its report.

4 Interface specifications

This chapter contains the specification for the unit operation interfaces in the CAPE-OPEN Interface System that were used to implement the prototypes demonstrated at the end of the project.

This specification is subject to the following assumptions and limitations:

- It supports the GRP1 sub-task of the Unit Operations work package and has been developed in the context of a simple mixer-splitter unit, with no external CAPE-OPEN numerical requirements.

- The host simulator is assumed to be sequential-modular and steady state.

Note: In this document, references to the term “component” refer to software components, rather than chemical species.

4.1 IDL definitions

This section contains the COM and CORBA IDL instructions. They are compilable files that you can use directly for producing CAPE-OPEN compliant components for Group 1 UNIT Operations, although, in the case of the CORBA files, we have not had the opportunity to test them fully.

4.1.1 COM IDL

Common CAPE-OPEN Definitions

```
#ifndef _FUNDAMENTAL_IDL_
#define _FUNDAMENTAL_IDL_
import "oaidl.idl";
import "ocidl.idl";


#define REALLYout in,out
// Include GUIDs
#include "COGuids.idl"

// Fundamental CAPE-OPEN data types
typedef long                  CapeLong;
typedef double                CapeDouble;
typedef VARIANT_BOOL          CapeBoolean;
typedef BSTR                  CapeString;
typedef VARIANT               CapeVariant;
#define CapeInterface LPDISPATCH
//typedef LPDISPATCH          CapeInterface;
typedef DATE                  CapeDate;

// Fundamental CAPE-OPEN array data types (automation
compatible)
typedef VARIANT               CapeArrayLong;
typedef VARIANT               CapeArrayDouble;
typedef VARIANT               CapeArrayBoolean;
typedef VARIANT               CapeArrayString;
typedef VARIANT               CapeArrayInterface;
typedef VARIANT               CapeArrayDate;

// Validation status values
[uuid(CapeValidationStatus_IID), version(1.0)]
typedef enum eCapeValidationStatus {
  CAPE_NOT_VALIDATED = 0,
  CAPE_INVALID = 1,
  CAPE_VALID = 2
} CapeValidationStatus;
```

```
// ICapeIdentification interface
// Provides methods to identify a CAPE-OPEN component.
[
      object,
      uuid(ICapeIdentification_IID),
      dual,
      helpstring("ICapeIdentification Interface"),
      pointer_default(unique)
]

interface ICapeIdentification : IDispatch
{
      // Get the name of the component
      //
      // CAPE-OPEN exceptions:
      // ECapeUnknown

      [propget, id(1), helpstring("property ComponentName")]
      HRESULT ComponentName([out, retval] CapeString *name);

      // Set the name of the component
      //
      // CAPE-OPEN exceptions:
      // ECapeUnknown, ECapeInvalidArgument

      [propput, id(1), helpstring("property ComponentName")]
      HRESULT ComponentName([in] CapeString name);

      // Get the description of the component
      //
      // CAPE-OPEN exceptions:
      // ECapeUnknown

      [propget, id(2), helpstring("property
ComponentDescription")]
      HRESULT ComponentDescription([out, retval] CapeString
*desc);

      // Set the description of the component
      //
      // CAPE-OPEN exceptions:
      // ECapeUnknown, ECapeInvalidArgument

      [propput, id(2), helpstring("property
ComponentDescription")]
      HRESULT ComponentDescription([in] CapeString desc);
};


// Typedef the interface to the base IDispatch pointer
typedef LPDISPATCH CapeIdentificationInterface;

#endif //_FUNDAMENTAL_IDL_
```

UNIT Specific Definitions

```
//Allowed directions for unit ports
[uuid(CapePortDirection_IID), version(1.0)]
typedef enum eCapePortDirection{
// changed from INPUT and OUTPUT to avoid clash with parameter
mode enumeration
      CAPE_INLET=0,
      CAPE_OUTLET=1,
      CAPE_INLET_OUTLET=2
} CapePortDirection;


//    Types of unit ports
[uuid(CapePortType_IID), version(1.0)]
typedef enum eCapePortType{
  CAPE_MATERIAL = 0,
  CAPE_ENERGY = 1,
  CAPE_INFORMATION = 2,
  CAPE_ANY = 3
} CapePortType;
```

ICapeUnit

```
//    This interface provides the basic functionality for a Unit
//    Operation component
[
      object,
      uuid(ICapeUnit_IID),
      dual,
      helpstring("ICapeUnit Interface"),
      pointer_default(unique)
]
interface ICapeUnit : IDispatch
{

//    Get the collection of unit operation ports
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown,  ECapeFailedInitialisation,  ECapeBadInvOrder

[propget, id(1), helpstring("Gets the whole list of ports")]
HRESULT ports([out, retval] CapeInterface* ports);

//    Get the collection of unit operation parameters
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown,  ECapeFailedInitialisation,  ECapeBadInvOrder,
//  ECapeBadCOParameter

[propget, id(2), helpstring("Gets the whole list of parameters")]
```

```
HRESULT parameters([out, retval] CapeInterface* parameters);

// Gets the flag to indicate unit's validation status
// notValidated(0),invalid(1) or valid(2)
//
// CAPE-OPEN exceptions
// ECapeUnknown, ECapeInvalidArgument

[propget, id(3), helpstring("Get the unit's validation status")]
HRESULT ValStatus([out, retval] CapeValidationStatus *valStatus);

//    Executes the necessary calculations involved in the unit
//    operation model
//
//  CAPE-OPEN exceptions raised:
//  ECapeUnknown,  ECapeBadInvOrder,  ECapeOutOfResources,
//  ECapeTimeOut,  ECapeSolvingError,  ECapeLicenceError

[id(4), helpstring("Performs unit calculations")]
HRESULT Calculate();

//    The unit operation is asked to read its persistent state,
//    from the storage location it has previously chosen in
//    Save.
//
//  CAPE-OPEN exceptions raised:
//  ECapeUnknown,  ECapeInvalidArgument,  ECapePersistenceNotFound,
//  ECapeIllegalAccess,  ECapeNoImpl

[id(5), helpstring("Recovers unit persistent state")]
HRESULT Restore([in] CapeVariant* storage);

//    The unit operation is asked to save its persistent state,
//    in a given location passed in as argument. This unit may
//    chose to use that storage (e.g. structured storage) or to
//    use its own internal storage mechanism (e.g. a plain ASCII
//    text file). If the storage location is changed by the
//    unit, the new location is sent back to the client.
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown,  ECapeInvalidArgument,  ECapePersistenceNotFound,
//  ECapeIllegalAccess,  ECapeNoImpl

[id(6), helpstring("Saves unit state")]
HRESULT Save([REALLYout] CapeVariant* storage);

//    The unit operation is asked to configure itself.Typically
//    some ports and parameters are created here
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown, ECapeLicenceError, ECapeFailedInitialisation,
ECapeOutOfResources

[id(7), helpstring("Configuration has to take place here")]
HRESULT Initialize();

//    Clean-up tasks can be performed here. References to
```

```
//    parameters and ports are released here.
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown, ECapeBadInvOrder

[id(8), helpstring("Clean up has to take place here")]
HRESULT Terminate();

//    Validate that the parameters and ports are all valid
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown,  ECapeBadCOParameter,  ECapeBadInvOrder

[id(9), helpstring("Validate the Unit")]
HRESULT Validate([REALLYout] CapeString* message, [out, retval]
CapeBoolean* isValid);

//    Set the simulation context
[propput, id(11), helpstring("Set the simulation context")]
HRESULT simulationContext([in] CapeInterface simContext);
};
```

ICapeUnitPort

```
//    This interface represents the behaviour of a Unit
//    Operation connection point (Unit Operation Port). Different
//    attributes for configuring the port as well as to connect
//    it to a material, energy or information
 [
      object,
      uuid(ICapeUnitPort_IID),
      dual,
      helpstring("ICapeUnitPort Interface"),
      pointer_default(unique)
 ]
 interface ICapeUnitPort: IDispatch
 {
 //    Returns the type of a this port (allowed types are among
 //    the ones included in the CapePortType type)
 //
 //  CAPE-OPEN exceptions:
 //  ECapeUnknown, ECapeFailedInitialisation

 [propget, id(1),  helpstring("type of port, e.g. material, energy or
 information")]
 HRESULT portType([out, retval] CapePortType* portType);

 //  Returns the direction in which the object connected to this
 //    port is expected to flow. Allowed values are among those
 //    included in the CapePortDirection type)
 //
 //  CAPE-OPEN exceptions:
```

```
//  ECapeUnknown, ECapeFailedInitialisation

[propget, id(2), helpstring("direction of port, e.g. input, output or
unspecified")]
HRESULT direction([out, retval] CapePortDirection* portDirection);

//   Returns to the client the object that is connected to this
//   port
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown, ECapeFailedInitialisation

[propget, id(3), helpstring("gets the objet connected to the port,
e.g. material, energy or information")]
HRESULT connectedObject([out, retval] CapeInterface*
connectedObject);

//   Connects an object to the port. For a material port it must be
//   an object implementing the ICapeThermoMaterialObject interface,
//   for Energy and Information ports it must be an object
//   implementing the ICapeParameter interface.
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown, ECapeInvalidArgument

[id(4), helpstring("connects the port to the object sent as argument,
e.g. material, energy or information")]
HRESULT Connect([in] CapeInterface objectToConnect,
                       [REALLYout] CapeString* message,
                       [out, retval] CapeBoolean* isOK);

//   Disconnects whatever object is connected to this port.
//
//  CAPE-OPEN exceptions:
//  ECapeUnknown

[id(5), helpstring("disconnects the port")]
HRESULT Disconnect([REALLYout] CapeString* message,
                          [out, retval] CapeBoolean* isOK);

};
```

ICapeUnitCollection

```
// This interface provides the behaviour for a read-only
// collection. It can be used for storing ports or
// parameters
[
      object,
      uuid(ICapeUnitCollection_IID),
      dual,
      helpstring("ICapeUnitCollection Interface"),
```

```
      pointer_default(unique)
]
interface ICapeUnitCollection : IDispatch
{

// Gets an specific item stored within the collection,
// identified by its name or index, that is passed in as
// an argument to the method.
[id(1), helpstring("gets an item specified by index or name")]
HRESULT Item([in] CapeVariant id,
             [out, retval] CapeInterface* item);

// Gets the number of items currently stored in the
// collection.
[id(2), helpstring("Number of items in the collection")]
HRESULT Count([out, retval] CapeLong* itemsCount);
};
```

ICapeUnitReport

```
// This interface provides access to the active unit
// report and the available list of options. It also
// allows to trigger the creation of a report
[
      object,
      uuid(ICapeUnitReport_IID),
      dual,
      helpstring("ICapeUnitReport Interface"),
      pointer_default(unique)
]
interface ICapeUnitReport: IDispatch {

//Get the list of possible reports.
[propget, id(1), helpstring("Gets the list of unit reports")]
HRESULT reports ([out, retval] CapeArrayString* reports);

// Get the current active report.
[propget, id(2), helpstring("Gets the active unit report")]
HRESULT selectedReport ([out, retval] CapeString* report);

// Set the active report equals to the [in] argument
[propput, id(2), helpstring("Sets a new active unit report")]
HRESULT selectedReport ([in] CapeString report);

// Produces the active report.
// Notice the [in, out] character of the argument message.
// This was added to allow the interface to be used in VB.
// Only [out] arguments do not seem to work very well in
// VB.
[id(3), helpstring("Creates the active report")]
```

```
HRESULT ProduceReport([in, out] CapeString* message,
                      [out, retval] CapeBoolean* isOK);
};
```

ICapeUnitEdit

```
//    This interface provides the editing functionality for a
Unit
//    Operation component should it provide one
[
      object,
      uuid(ICapeUnitEdit_IID),
      dual,
      helpstring("ICapeUnitEdit Interface"),
      pointer_default(unique)
]
interface ICapeUnitEdit : IDispatch
{
      //    Displays the unit graphic interface, if available.
      //
      // CAPE-OPEN exceptions:
      // ECapeNoImpl, ECapeBadInvOrder

      [id(1), helpstring("Displays the graphic interface")]
      HRESULT Edit([REALLYout] CapeString* message,
                      [out, retval] CapeBoolean* isOK);
};
```

ICapeUnitPortVariables

```
[
      object,
      uuid(ICapeUnitPortVariables_IID),
      dual,
      helpstring("ICapeUnitPortVariables Interface"),
      pointer_default(unique)
]
interface ICapeUnitPortVariables: IDispatch
{

//  Get the position of a port variable in the EO model - used to
//  correctly build the equations representing a connection to this
//  port.  Variable type can be - flowrate, temperature, pressure, //
//  specificenthalpy, VaporFraction and for Vapour fraction component
//  name must also be specified.
//
//  CAPE-OPEN exceptions:
//

[propget, id(1),  helpstring("Return index of port variable in EO
Model given its type")]
```

```
HRESULT Variable([in] CapeString Variable_type, [in] CapeString
Component,[out, retval] int* index);

// Set the position of port variables: this should ultimately be a
// private member function

[id(2), helpstring("Set index of port variable in EO model given its
type")]
HRESULT SetIndex([in] CapeString Variable_type, [in] CapeString
Component,[in] int index);

};
```

4.1.2 CORBA IDL

```
// ---- The scope of the Unit Operations specification
// Reference document: http://www.global-cape-
open.org/05_CO_Unit_Operations.pdf
module Unit {

// Forward declaration of interfaces
interface ICapeUnitManager;
interface ICapeUnit;
interface ICapeUnitReport;
interface ICapeUnitPortCollection;
interface ICapeUnitParameterCollection;
interface ICapeUnitPort;

// Interface sequence
typedef sequence<ICapeUnitManager> CapeArrayUnitManager;
typedef sequence<ICapeUnit> CapeArrayUnit;
typedef sequence<ICapeUnitReport> CapeArrayUnitReport;
typedef sequence<ICapeUnitPortCollection>
CapeArrayUnitPortCollection;

typedef sequence<ICapeUnitParameterCollection>
CapeArrayUnitParameterCollection;
typedef sequence<ICapeUnitPort> CapeArrayUnitPort;

enum CapePortType{
      CAPE_MATERIAL,
      CAPE_ENERGY,
      CAPE_INFORMATION,
      CAPE_ANY
};
enum CapePortDirection{
      CAPE_INLET,
      CAPE_OUTLET,
      CAPE_INLET_OUTLET
};
```

```
enum CapeUnitValStatus{
      CAPE_NOT_VALIDATED_UNIT,
      CAPE_INVALID_UNIT,
      CAPE_VALID_UNIT
};

typedef sequence<CapePortType> CapeArrayPortType;
typedef sequence<CapePortDirection> CapeArrayPortDirection;
typedef sequence<CapeUnitValStatus> CapeArrayUnitValStatus;

interface ICapeUnitManager :
   Common::Identification::ICapeIdentification {
ICapeUnit CreateUnit()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeOutOfResources);
void Shutdown()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeNoImpl);
};

interface ICapeUnit :
   Common::Identification::ICapeIdentification {

void Initialize ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeLicenceError,
           Common::Error::ECapeFailedInitialisation,
           Common::Error::ECapeOutOfResources);

void Terminate ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeBadInvOrder);

void Calculate ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeBadInvOrder,
           Common::Error::ECapeOutOfResources,
           Common::Error::ECapeTimeOut,
           Common::Error::ECapeSolvingError,
           Common::Error::ECapeLicenceError);

ICapeUnitPortCollection GetPorts()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation,
           Common::Error::ECapeBadInvOrder);

ICapeUnitParameterCollection GetParameters ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation,
           Common::Error::ECapeBadInvOrder,
           Common::Error::ECapeBadCOParameter);

Base::CapeBoolean Validate(out Base::CapeString message)
```

```
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeBadCOParameter,
           Common::Error::ECapeBadInvOrder);

CapeUnitValStatus GetValStatus()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeBadInvOrder);

ICapeUnitReport GetReportObject ()
   raises (Common::Error::ECapeUnknown);

void Save(in Base::CapeString storage)
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument,
           Common::Error::ECapePersistenceNotFound,
           Common::Error::ECapeIllegalAccess,
           Common::Error::ECapeNoImpl);

void Restore(in Base::CapeString storage)
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument,
           Common::Error::ECapePersistenceNotFound,
           Common::Error::ECapeIllegalAccess,
           Common::Error::ECapeNoImpl);

Base::CapeBoolean Destroy()
   raises Common::Error::ECapeUnknown);

};

interface ICapeUnitReport :
   Common::Identification::ICapeIdentification {
Base::CapeArrayString GetReports ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeNoImpl);
Base::CapeString GetSelectedReport()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeNoImpl);
void SetSelectedReport(in Base::CapeString report)
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument,
           Common::Error::ECapeNoImpl);
Base::CapeString ProduceReport ()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeNoImpl);
};

interface ICapeUnitPortCollection :
   Common::Identification::ICapeIdentification {
Base::CapeLong Count()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation);
ICapeUnitPort Item(in Base::CapeLong index)
```

```
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument,
           Common::Error::ECapeFailedInitialisation,
           Common::Error::ECapeOutOfBounds);
};

interface ICapeUnitParameterCollection :
   Common::Identification::ICapeIdentification {

Base::CapeLong Count()
  raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation);

Common::Parameter::ICapeParameter Item(in Base::CapeLong
index)
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument,
           Common::Error::ECapeFailedInitialisation,
           Common::Error::ECapeOutOfBounds);
};

interface ICapeUnitPort :
   Common::Identification::ICapeIdentification {

CapePortType GetType()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation);

CapePortDirection GetDirection()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation);

Thrm::Cose::ICapeThermoMaterialObject GetConnectedObject()
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeFailedInitialisation);

void Connect(in Thrm::Cose::ICapeThermoMaterialObject matObj)
   raises (Common::Error::ECapeUnknown,
           Common::Error::ECapeInvalidArgument);

void Disconnect()
   raises (Common::Error::ECapeUnknown);
};

}; // END Unit module
```

5 Notes on analysis and interface specifications

The purpose of these notes is to record the rationale for the decisions made in designing the Unit Operations interfaces, methods and attributes and to indicate those issues that require further investigation.

5.1 Issues to be resolved

The following issues have been identified, but not yet fully resolved. Experience gained in analysing the implementation of the UNIT prototypes will be a major factor in their resolution.

- Reporting, especially the supported reporting styles and devices, is not fully defined as yet. The prototype supports a simple report.
- Energy and Information objects (not required for the prototype).
- Persistence of a unit's data (not required for the prototype)
- Calculation progress monitoring and interrupts (not required for the prototype).

5.2 Decision rationale

5.2.1 Whenever a unit is used there must be a check on its validity.

The reason for this is to check that the plug-in unit is indeed a CAPE-OPEN compliant unit, rather than the movie player, or some other ActiveX control. There should be three levels of check: the label on the box should say it has passed the validation suite; the unit must support the required interfaces and the interfaces must have the required behaviour.

The registration process could do a validation test. We must be careful not to test validity in such a way as to force a particular architecture. Some interfaces may be on components such as port collections, which are constructed at run time by the unit. We should only test for interfaces that are guaranteed to be present before the unit has done anything.

It is important to check the unit validity every time we want to use it because the registry can be corrupted in between registration and different sessions of usage.

5.2.2 The three types of port are to be merged into one type.

The three types of port: material, energy and information, have a lot of functionality in common. By combining the three into one we can simplify the interface to a useful degree. Each port type is to be distinguished by the value of an attribute.

5.2.3 The CO Ports interface will be restricted

The ports collection will not have Add and Remove methods because:

- It is desirable to provide a client direct access to a unit's ports.
- In order to recognise the unit's responsibility for managing its own ports, we restrict what someone else can do to the collection by removing the power to add and delete ports. Only the unit can do these actions.
- We do not want a client to add ports because the unit would not know about the addition.
- If a unit gives a collection to another client (e.g. its ports) there is a danger of crashes. If the client were able to delete a port and the unit then tried to access that now non-existent port through a now invalid reference stored in its own copy of the collection, nasty things could happen.

5.2.4 Ports have direction

It is an assumption that ports can have a specified direction that refers to input or output. This direction reflects the expected flow of e.g. materials to or from the Unit Operation, but does not imply or restrict the actual flow direction.

We make the stipulation that:

- An input information port can only carry an independent (input) parameter.
- An output information port can only carry a dependent (calculated) parameter.

This allows us to treat information, material and energy ports in a similar way.

5.2.5 Parameters

CAPE-OPEN parameter interfaces are now defined as a CAPE-OPEN Common Service. Details of the proposed interfaces can be found in the document "Parameter Common Interfaces".

Previous versions of this document describe a set of interfaces for parameters that are implemented in the 0.9 version of the CAPE-OPEN IDL. Simulators that are compatible with the 0.9 IDL will support these, unofficial, parameter interfaces.

Support for the official parameter interfaces is included in the 0.9-3 IDL. Simulators that are compatible with the 0.9-3 specification will support the latest parameter interfaces.

Unit operation developers should work to the 0.9-3 specification to save doing rework.

5.2.6 User Interfaces

There has been some debate on the role of the User Interface with respect to a unit. Some documents declare that the unit does not have one, others do. We decided that:

- The unit may or may not be provided with its own UI.
- The simulator host invokes the unit's view method to start it up. It the unit has one, it displays it and allows the user to enter/edit the unit's data. If it does not, it returns a suitable error. The host can then use this error to decide what to do next. Typically, the host can query the unit for the list of its input parameters and build a UI from that. Once the user has closed the generated UI, the host resets the parameters.
- If the unit is run from another simulator, the unit is allowed to invoke that simulator's UI, but only for data entry and editing.
- The host handles the flowsheet topology and connections. The unit is told what streams are to be connected by the host.
- Units working remotely from the host will not be able to display a UI, so the host will be required to provide one. A possible solution to this is the use of web technology. More precisely, it would possible for a remote client to display the UI of its server component, although in many situations this could appear quite strange for the human user. This point should be studied in later project stages.

Notes on the Edit capability of the ICapeUnit interface

It has been pointed out that the displaying of graphical user interfaces is not handled in CORBA in the same way as in COM. In addition it has been also pointed out that the method

Edit in the interface ICapeUnit would not have much sense either in distributed COM environments.

Two solutions were considered that focus on:

- A clear and correct usage of interfaces
- An improved design.

The first solution simply considers that clients of the interface ICapeUnit will do their job intelligently and thus they will invoke the method call just when it makes sense to do it (e.g. when client and server both run in the same memory space or process).

The second solution modifies the design of the interface ICapeUnit by again taking advantage of factorization. It proposes to move the method Edit to a new interface (e.g. ICapeUnitEdit) that, similarly to ICapeUnitReport, would need to be supported only by those unit operations that require a graphical user interface.

The second solution was adopted for the reasons that it provides a higher degree of compatibility with CORBA environments and that it avoids all clients have to implement the method even if they do not need it.

5.2.7 Undo

UI's impose a requirement for an undo facility. This arises because, when the user closes the generated UI by pressing the OK button, the data must be validated somehow. Only the unit has the knowledge to do this, so the updated values must be copied back to the unit's parameters and Validate() called. The problem comes if the data fails validation and the user decides to forget the changes and presses the cancel button. The unit must then revert back to the original data that was present before the UI was displayed.

Because undo is a fairly involved facility to implement (if any number of undos are required) it was decided that undo should be handled entirely by the Simulator Executive. Whenever some of the values of the Public Unit Parameters are going to be changed, the simulator caches the original values, thus, if validation fails, the simulator puts back the original values.

5.2.8 Persistence

- We assume that the prototype unit is able to store its data in a file separate from that of the main simulation case file (for the first prototype). This is because CAPE-OPEN persistence mechanisms have to be studied carefully while prototyping.
- The save method needs to return the name of the file used to store its data so that the host can store it with the rest of the simulation case.
- The persistence of the flowsheet connections to the Flowsheet Unit is the responsibility of the host simulator only. When a case is loaded into the host, it reads the connectivity and connects each instance of the unit with appropriate material, energy, etc objects. The Flowsheet Unit does not need to know anything about connectivity.

- Testing and the development of example Flowsheet Units have revealed two problems with the Save and Restore methods defined as part of the ICapeUnit interface:

  - It is not possible to provide support for passing an IStorage pointer into a Visual Basic component on the Save and Restore calls, because VB does not provide support for using the IStorage interface

  - Without being able to use the IStorage option, a VB Flowsheet Unit can only write its data to an independent file. The result of this is likely to be large numbers of loose files that are at risk of being deleted or lost when simulation data is moved or 'tidied up'.

- To address these problems it is recommended that COSEs that support COM Flowsheet Units should support the standard COM persistence interfaces in addition to, and in preference to, the CAPE-OPEN Save and Restore methods.

- For COM implementations of CAPE-OPEN Flowsheet Units it makes sense to support standard COM persistence mechanisms. In particular, setting the 'persistable' property of a Visual Basic component to true, means that it will automatically support the IPersistStreamInit interface and the developer simply has to fill in the necessary methods to write the component's data to a Visual Basic PropertyBag.

- In general, to support COM persistence a component must implement one of three possible interfaces: IPersistStorage, IPersistStream or IPersistStreamInit. Depending on which is supported the COSE will pass an IStorage or IStream interface to the unit which it can use to persist its data. (Note IStorage corresponds to a directory and IStream to a file, a storage can be populated with other storages and streams, and given the IStream interface of a stream within a storage it is possible to read and write data in any format.)

- For Flowsheet Units that do save data to loose files no assumptions should be made about the existence or lifetime of the file. It is safest to assume that the file only exists while the unit operation is writing to it or reading from it. This assumption allows a COSE to manage the files itself. For example a COSE that supports COM interfaces could copy the contents of a file to an internal stream once the Flowsheet Unit's Save operation has completed. The COSE could then delete the file. To restore the data the COSE could recreate the file (possibly with a different name and in a different location) and then call the unit's Restore method passing in the name of the new file. This solution means that the user does not have to worry about loose files accumulating, while the Flowsheet Unit author need only implement the ICapeUnit Save and Restore methods. Note that this solution is only possible if the Flowsheet Unit writes its data to a file on a file system to which the COSE has access. This would probably not be the case if the Flowsheet Unit were executing on a different machine to the COSE.

- For CORBA implementations there is no standard persistence mechanism so it is not clear how the problems of the Save and Restore methods should be addressed. A future version of this document will address this issue when more experience of implementing CORBA Flowsheet Units is available.

5.2.9 Reporting

- It is assumed that the host is able to handle unit reporting by asking the unit to produce its own style of reports and also by querying the unit for its calculated parameters.

- The most basic style of report is a plain ASCII text file, which can be displayed by the host. There may need to be a method to return the name of this file if the host is to make use of it.

- If the unit has its own UI, it is at liberty to display its reports in any way that it wants (e.g. graphics, tables etc.).

- Because a unit may have multiple styles of report (e.g. brief, detailed), we provide a method for the host to inquire what reports are available. The user can select a report from this list and the host tells the unit which one to do.

5.2.10 Argument Typing

There was some discussion on the need or otherwise for the strong typing of method arguments.

The arguments for strong typing were:

- Strong typing leads to fewer errors because the compiler enforces argument type matching.

- No casting is required so the component transfers its arguments quicker.

The alternative is that it makes better sense to return pointers to IDispatch interfaces for certain types of argument because:

- It is more flexible.

- It is more future proof because it allows upgrading to newer interfaces. For example, GetPorts returns an IDispatch for an ICapeUnitPorts interface. This would allow a change to a future ICapeUnitPorts2 interface if one becomes available.

- It is compatible with variant data types.

The conclusion was to use strong typing where flexibility would not be decreased and to use IDispatch where appropriate. It is expected that the experience gained in building the prototype will shed more light on what is needed.

Another question that arose was whether to allow method overloading. It was decided that it was simplest to allow variant arguments for this rather than multiple arguments of different types.

5.2.11 Tracing and Interrupts

It was decided that units should implement a tracing capability so where units take a long time to calculate, some progress monitoring or diagnostic output can be done. This would be made possible by passing to the unit a pointer to some sort of simulator context object, which it could use to send the monitoring information. This has yet to be defined.

It was also realised that this would allow interrupts to be done because the unit could be required to poll the simulator context at regular intervals. A flag in the context would indicate if the unit were to continue or not.

It was realised that there would be no need for a resume method to restart an interrupted calculation, because the Calculate method could easily handle this by remembering if its deliberations were interrupted or not. If they were, then a call to calculate after an interrupt is the same as a call to resume. Prototyping work will discover the necessary arguments for this method.

5.2.12 Termination

There was some debate on if the unit needs a terminate method. It was decided that it did and would typically carry out the release of resources allocated during the unit's initialisation. Terminate would also need an argument to indicate if the termination was "soft" or "hard".

- Soft means that it checks to see if the unit's data needs to be saved and, if so, return a suitable error to the host. The host can ask the user if the data is to be saved and if so, call the save method.

- Hard means that all errors occurring in the termination, including unsaved data, are to be ignored. The method must be intelligent and not attempt to free resources more than once. This might be required if terminate fails for some reason (e.g. a resource might be locked by another process and cannot be freed). The host might ask if the user wants to try to terminate again.

We added an attribute "dirty" to ICapeUnit which is true if the data is dirty (i.e. changed but not saved), so that hosts can detect if the data needs to be saved.

5.2.13 Initialisation

The initialisation of a unit requires information about not only the simulator context mentioned earlier, but also references to the thermodynamic and numerical servers to be used. The problem that arose was where to pass this information. If it were passed during initialisation, there was the danger that it could be invalid by the time the unit wanted to do its calculation. If it were passed during calculation it would be unavailable during initialisation. It was decided to pass this information in the simulatorContext, which would become an attribute of ICapeUnit. It could then be set at any time, rather than having to force the client to do it at one prescribed time.

Initialize, CheckPorts and CheckParameters

From the viewpoint of a neutral reviewer of the interface, the following general comment was raised:

The three methods above require more explanation on topics such as, the operations that need to be performed to initialize a Unit Operation, or when the client should request a Unit Operation to check its ports and variables.

This raises questions like:

- Are ports and variables created and initialized within Initialize? Is the simulation context accessed inside this method? Are internal material objects created here?
- Or, in CheckPorts, is it checked that essential ports and non-essential ports are connected? If a non-essential port is not connected is this an error?
- Or, what exactly CheckParameters does? Does it allow for parameters that can be both inputs and calculated values?

It was agreed that these matters will be clarified by the prototyping activity, and therefore will be incorporated in the specification after the prototypes have been fully evaluated. In the short-term, it is proposed that these checks will be replaced by a Validate method, until the prototyping can define their individual behaviours better.

5.2.14 Connecting Streams

Initially, the process of connecting streams was handled by just assigning a reference to a material, energy or information object to the connectedObject attribute of a port. It was decided that this was a bit obscure, so connect and disconnect methods were specified. These methods could handle error checking, if required. For example, they could check that a material port was being connected to a material object. An information port could check the dimensionality of the parameters at each end of the connection.

5.2.15 Adding and Removing Ports

This section summarises the pros and cons of the different alternatives for adding and removing ports.

Scenario 1

The first design we came up with considered ports grouped in collections that exposed a regular collection interface (e.g. `Add`, `Remove`, `Item` and `Count`). In this approach, creation of ports was requested directly from the collection. Thus, the simulator got access to the collection of ports by using `getPorts([in] filter)` (note that filter represents a group of strings specifying the type of ports wanted) and after that invoked the method `::Add()` to create the new port.

The first trade off we found relates to the association between the collection of ports and the unit operation. In principle, the client could ask for a collection of ports filtered in such a way that it did not mirror the way in which the unit operation internally stores its ports (e.g. the client could ask for all ports, while the operation classifies them in input and output). The result

was that the unit operation had to be able to create "on-the-fly" a new collection to be delivered to the client.

This implied that the unit operation had to be notified by the different "temporary" collections, when a new port was being added, in order that the unit operation could update its internal collections to reflect the new status. Similarly when the client requested to delete an existing port, the unit operation had the right of checking whether that port was allowed to be deleted or not (e.g. in some circumstances the unit operation can decide to lock some ports that it considers as essential ports for its configuration). An internal mechanism for the unit operation to flag ports as removable or non-removable (e.g. a private interface in every port) was therefore necessary.

Scenario 2

Moving the methods `AddPort` and `DeletePort` to the `ICapeUnit` interface was intended to recognise the right of every unit operation to be totally responsible for the creation or removal of ports, rather than putting some other ancillary components (e.g. collections of ports) in charge of this activity. In this approach, the unit operation can easily determine whether to create or not a new port and to delete it or not, based on its private information (e.g. maximum number of ports of a given type, or essential ports vs. non-essential ports).

With this second approach the collection of ports becomes a read-only collection (no items can be added or removed) and no change in the configuration of a unit operation can be done without asking the unit operation directly.

Nevertheless, this second scenario does not solve the problem about the `RemovePort` method either. As well as in the first scenario, clients such as the Simulator Executive should not ask for the deletion of a port if they did not ask for its creation beforehand. Only the creator of a port has the right to delete it, otherwise the method should fail in order to avoid the original configuration of the unit operation being lost. We envision that, during the initialisation stage, the unit operation may add some "essential ports", whose deletion may be forbidden.

Scenario 3

Basically, the need of adding new ports comes from the idea of designing a flexible unit operation, whose number of connection points can be extended at run-time. But, rather than extending the number of connection points, what is really important is to be able to extend the number of actual connections.

From this point of view the client of a unit operation may be disappointed by the fact that it has to create a new port explicitly to perform a stream connection. We thought about the possibility of redesigning the methods `AddPort` and `RemovePort` to be `Connect` and `Disconnect`. `Connect` would be allowed to pass a stream object (e.g. material, information or energy) along with the type of connection wanted to be performed (e.g. `"material", "input"`). Therefore creation and deletion of ports would be hidden inside the unit operation.

Two disadvantages were detected: I) Two separate actions take place in a single call to `Connect` (e.g. creation of the port and its connection to a given stream), that somehow hides the source of possible errors when the method fails. The same also applies to `Disconnect`.

II) The existence of two `Connect` methods (in `ICapeUnit` and `ICapeUnitPort`) may confuse clients of the interface.

Scenario 4

This scenario emphasises more the idea of keeping the design of interfaces as simple as possible.

From that viewpoint, reconfiguration of unit operations by e.g. adding new ports or removing some of the existing ones, has to be the responsibility of the unit operation itself, in the same way as the initial configuration is also responsibility of the unit operation (e.g. addition of essential ports).

In a possible architectural solution, a collection of ports can be intelligent enough to extend itself by replicating an item (a port), when that item is going to be connected. Similarly, when a port is disconnected, the collection may decide just to disconnect it, or to disconnect and remove it, depending on the information that its parent unit operation has provided to configure the collection (e.g. is the port essential or non-essential?). It is clear that a private interface between the collection of ports and the unit operation is necessary also here.

Some unit operations may be coded in such a way that their "essential ports" do not replicate themselves when one is connected, while others, e.g. "non-essential ports", do. This information also has to be passed to the respective ports by using private means only known by the unit operation. Similarly, some units may want to impose a certain limit in the number of allowed ports of a given type. Therefore this information needs also to be passed to the collection of ports.

Scenario 5

This scenario takes advantage of factorization of interfaces and the fact that a single object can support multiple interfaces in order to customize its functionality as desired. This proposal considers the existence of a new interface named IPortAdministration that contained the methods AddPort and RemovePort. Simple Unit Operations that do not contain non-essential ports (e.g. possibly a pump) would not need to support this interface, while flexible unit operations, such as a distillation column, would support it.

Final consideration

Finally, we decided to propose an interface design in which the purpose of every method is clear and concise, e.g. every method serves to perform one action (enhancement of the unit operation is separated from its actual connection to the flowsheet). The price was to give some extra work to the client that has to perform actions such as `AddPort` or `RemovePort`.

Conclusion

In our opinion the differences between the different approaches concern the complexity of necessary "backdoors" that the unit operation is going to use, in order to control its collection of ports appropriately. The attractive thing of the fourth approach is the simplicity of the interfaces, while the attractive thing of some of the others is the ease of implementation. All of them are perfectly valid designs.

We cannot know all possible configurations required by future unit operations. We realised that there could be additional non-CAPE-OPEN functionality provided by the vendor of the unit. For example, the unit could allow the maximum number of ports to be variable, or allow port deletion. The AddPort and RemovePort methods on the old ICapeUnitPorts interface were therefore removed because the Simulator Executive no longer needed them.

This allows the Unit Operation to implement the Port and Port collections, get tied to those implemented by a third party vendor, or choose the ones provided by the Simulation Executive hosting the Unit Operation. The important thing here is that the interface between the Port / Ports collection and Simulator Executive follows strictly the standard CAPE-OPEN. So we considered a scenario in which Port and Ports could be implemented anywhere.

The final decision was to use Scenario 4 for now, in order to simplify the interfaces as much as possible. The other scenarios will be considered as prototyping progresses to a stage when these port issues become more pressing, such as for a more complex unit operation.

5.2.16 The Relationship of UNIT with THRM

This section gives more background on how the Unit Operation and the thermodynamic pieces of CAPE-OPEN are coupled together. This coupling involves the use of the MaterialPort and the MaterialObject concepts. In addition, some Unit Operations may require MaterialObjects to be created from MaterialTemplates not associated with any MaterialPort. The concept of a NonMaterialPort is introduced, as a means of delivering these additional MaterialObjects to the Unit Operation.

Below is the UML diagram, which shows the major classes and their relationships from an object modelling perspective. Not all the methods for these classes are indicated, as this would tend to distract from the main purpose of this diagram.

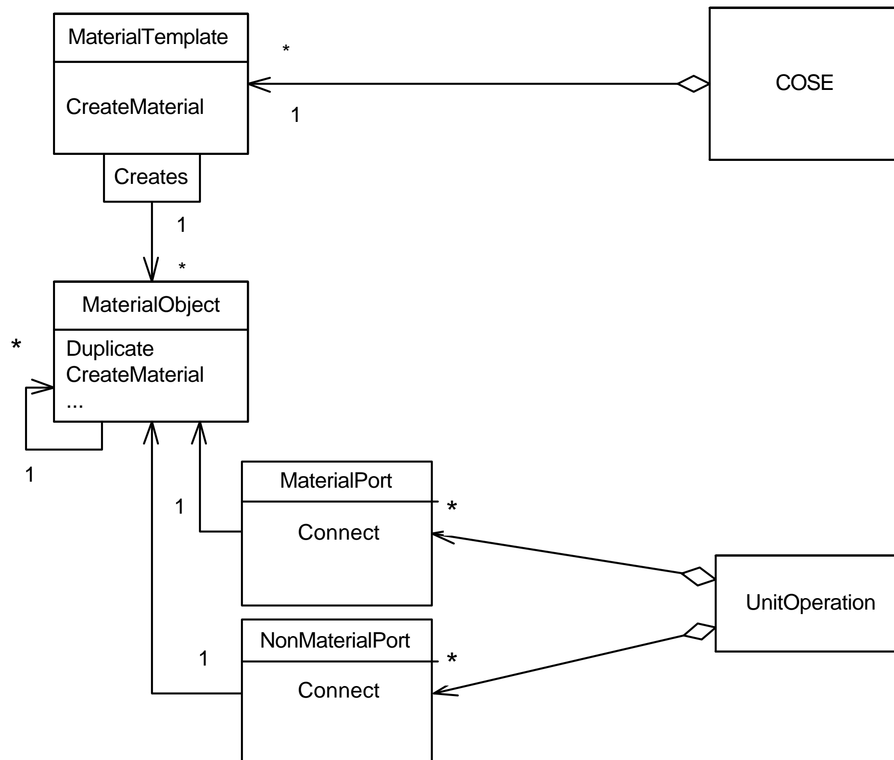


Figure 5.1 Unit & Thermodynamic Relationships

So to summarise the diagram:

A Unit Operation *has* a number of MaterialPorts and NonMaterialPorts

- A MaterialPort *is associated with a single* MaterialObject
- A NonMaterialPort *is associated with a single* MaterialObject
- This provides the means for the UnitOperation to perform thermodynamic property calculations on the MaterialObject
- A COSE *has a number of* MaterialTemplates
- A MaterialTemplate *creates a number of* MaterialObjects
- A MaterialObject *may Duplicate itself* or *Create a new* MaterialObject from its Creator MaterialTemplate

As far as interfaces are concerned, this agreed position adds two additional methods to the `ICapeThermoMaterialObject` interface (`Duplicate()` and `Create()`). A new interface, the ICapeNonMaterialPort, is also introduced to provide access to specific MaterialObjects (Templates) not associated with streams entering or leaving the units. This interface replaces the ICapeSimulationContext proposed previously.

The disadvantage of the SimulationContext approach was that it delivered all the Material Templates. Further data and computation would be required for a unit model to ascertain the particular Material Object required by the user and to employ it.

As a by-product of this revision, we have eliminated direct access between MaterialTemplates and user models. It now becomes unnecessary to provide CAPE-OPEN standard interfaces to Material Templates. The actual saving may be marginal, because the Objects mostly reproduce the behaviours required of the Templates.

6 Prototype implementation

This section has been removed since there are now working commercial examples that illustrate how to use the interfaces described in this specification.

7 Glossary

The terms used in this document are in accordance with the CAPE-OPEN Project Glossary[4].

8 References

1. **Brite/EuRam project BE-3512 CAPE-OPEN Annex 1 Project Programme, section 2.4.3**

2. **CAPE-OPEN Concepts**

3. **CAPE-OPEN Methods & Tools**

4. **CAPE-OPEN Glossary**

There are further general references in the **CAPE-OPEN Final Report**.

9 Appendix 1: Equation-Oriented Simulation

9.1 Introduction

In this appendix, we consider the extension of the Unit Operations interfaces to handle equation-oriented simulation, both steady-state and dynamic. This represents the GRP2 sub-task of the UNIT work programme.

This analysis was developed in close collaboration with the Numerical work package and confirmed that only minor extensions would be needed to the work already done by both work packages. However, time has not permitted us to build a prototype demonstration, which is why this chapter is contained in an appendix.

The chapter summarises an analysis of the equation-oriented approach, maps it onto existing Unit Operations and Numerical concepts and then presents a modified version of the mixer/splitter scenario shown previously in the main body of the document.

The analysis provides an overview of the types of equations set up by a typical EO unit, the functionality of the Equation Set Object (ESO) defined by the Numerical work-package (NUMR) and the way that an EO simulator brings together the individual unit operations to create a complete flowsheet.

9.2 An Equation-oriented Unit Operation

This appendix is developed around a simple unit operation, such as the mixer/splitter example used throughout this document. As before, this provides good insights into the issues involved, without becoming distracted by the complexities of the unit operation itself.

9.2.1 Equations for a typical EO unit operation

For a simple tank, with an inlet flow, and an outlet flow governed by the liquid head, then a typical set of equations would be:

$$\begin{aligned}
\frac{dM}{dt} &= F_{in} - F_{out} \\
\rho g h &= \frac{4 f L v^2}{2D} \\
\rho A h - M &= 0 \\
F_{out} &= \rho \frac{\pi D^2}{4} v
\end{aligned}$$

for laminar flow

$$f = \frac{16}{\mathrm{Re}}$$

for turbulent flow:

$$\frac{1}{\sqrt{f}} = -4\log_{10}\left(\frac{1}{\mathrm{Re}\sqrt{f}} + \ldots\right)$$

with the switch from laminar to turbulent flow occurring at Re=2100, and the reverse at Re=2000.

9.2.2 The Equation Set Object (ESO)

All the features of this set of equations, including the discontinuity in the equation for the friction factor, *f*, can be handled by the ESO defined by NUMR. The ESO allows the description of a set of equations:

$$f(x, \dot{x}, t) = 0$$

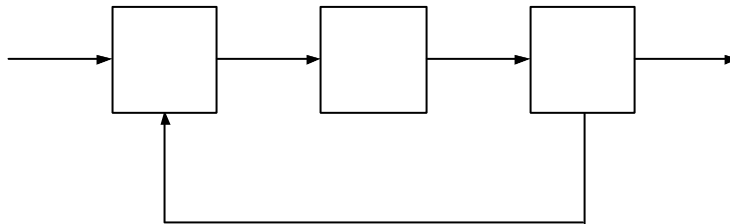
and provides methods for (amongst others):

- Number of variables
- Number of equations
- Get / set values of x, $\dot{x}$ and t
- Get values of functions
- Get number of non-zero derivatives
- Get structure of derivatives (Jacobian)
- Get / Set values of derivatives

Note that values of x are simply stored as a vector, individual values are accessed using the location within the vector.

9.2.3 Operation of an EO simulator

For the simple flowsheet below:



an EO simulator would generate the overall set of variables and equations by looking at each unit operation in turn to identify the variables and equations associated with the units themselves, then would add the connectivity equations and the user specification equations.

For example, in an EO simulator (here assumed to be working in steady-state, but extendible to dynamics) a single Unit Operation (UO) will describe a set of equations

$$\underline{f}(\underline{x}) = \underline{0}$$

where $\underline{x}$ is the vector of all the variables associated with the UO. This vector can be sub-divided into 3 categories:

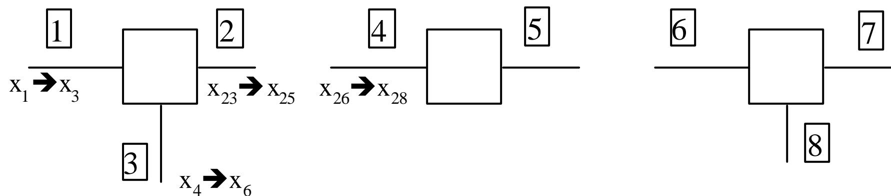
1. The 'input' variables $\underline{u}$. These would include variables such as feed stream data (global variables) and unit input data (Public Unit Parameters).

2. The 'output' variables $\underline{y}$. These would include variables such as product stream data (global variables) and calculated results that are made available to the simulator (Public Unit parameters).

3. Internal variables $\underline{x}'$, not visible to the Simulator Executive, other than as an element of the vector $\underline{x}$.

Thus $\underline{x} = \left[\underline{u}, \underline{x}', \underline{y}\right]$. Note again that the ESO simply stores $\underline{x}$ as a vector, and there is no information stored as to the type or meaning of a given element of the vector. Furthermore, in general for a single UO there will be fewer equations than there are variables, as many of the equations involving the input variables $\underline{u}$ can only be defined at the flowsheet level.

Returning to our simple flowsheet, let's assume that each stream has 3 variables associated with it and that each unit operation has just 1 item of input data. Let's further assume that the first unit operation has a total of 25 variables (x_1 to x_{25}). Of these, x_1 to x_6 are the variables associated with the feed streams, x_{23} to x_{25} are the variables associated with the product stream, whilst x_{10} is the UO input data. Finally, the UO itself is described by a total of 18 equations. A similar analysis can be performed for each of the other UO's in the flowsheet. Let's say that the 2nd UO has a total of 15 variables and 11 equations, whilst the 3rd has 40 variables and 36 equations.



A typical EO simulator will obtain this information from the ESO and will construct variable and equation arrays for the overall flowsheet:

Variables	1-25	26-40	41-80	
Equations	1-18	19-29	30-65	66-80

Thus, in this case, the simulator will add a further 15 equations (equations 66-80) to describe the overall flowsheet connectivity and the unit input data. These equations are derived from:

stream 1 = plant feed stream data (e.g. x_1 = value)

stream 4 = stream 2 (e.g. $x_{26} = x_{23}$)

stream 6 = stream 5

stream 3 = stream 8

and

x_j = UO input data (e.g. x_{10} = value)

where there are 3 equations for each stream and 1 for the input data for each UO.

9.2.4 GRP2 proposal

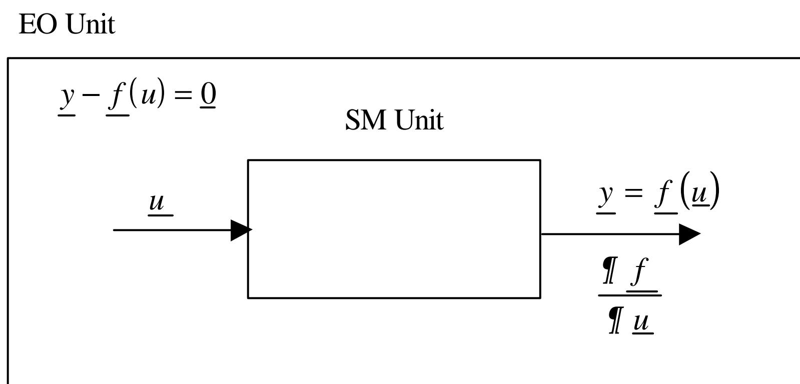
It is proposed that the variables and equations for each unit operation can be described by a ESO generated by that UO. Furthermore, the Simulator Executive will be able to generate an overall ESO by amalgamating the individual UO ESO's with the additional equations required to describe the flowsheet connectivity and UO input data. However, as each individual ESO contains a simple vector representation of the variables for the UO, methods must be available to determine the association of the UO variables with the flowsheet streams and with the UO input data.

Thus an EO unit will consist of the following:

- An ESO generated by the Unit operation, but exposed via the standard ESO interfaces.
- Methods to allow the user to provide UO input data (as per current SM UO interfaces)
- Methods to allow production of reports (as per current SM UO interfaces)
- Methods to allow the association of an internal ESO variable to a Global Variable (from the Material Object).
- Methods to allow the association of an internal ESO variable to a Public Unit Parameter. Note that all internal UO variables which can be specified by the user must be available as PUP's. Other internal variables are at the discretion of the UO developer.

9.2.5 Interoperability of SM and EO unit operations

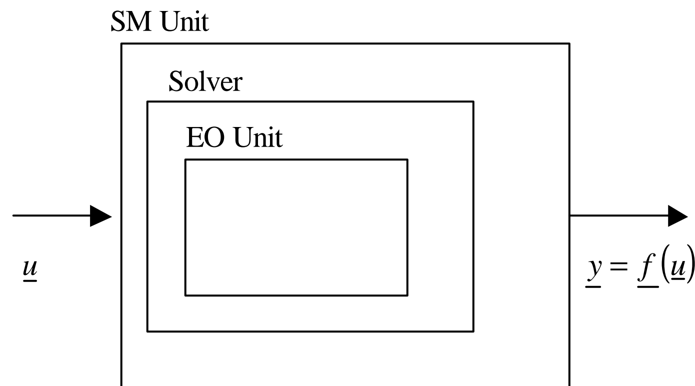
A CAPE-OPEN compliant SM unit operation can be used in an EO simulator as follows:



This implies that the SM CAPE-OPEN interface specification must make provisions for the derivatives to be accessed, if available. This is a new method, which is not covered by the current specification. Of course, individual implementations of such interfaces may choose not to provide such derivative information; in this case, the EO wrapper object will be unable to provide derivatives to its clients.

Furthermore, a legacy SM Unit operation can be used within an EO simulator by first wrapping as a CAPE-OPEN compliant SM unit operation, then using the method above.

Using a CAPE-OPEN compliant EO Unit Operation with a SM simulator is also possible, as below:



9.2.6 Other issues

EO thermodynamics and sub-flowsheets can both be handled as sub-ESO's, which can then be amalgamated into the overall flowsheet ESO.

The requirement to be able to identify the name of each of the variables within an ESO has been identified. While this is not strictly necessary mathematically, it enables a solver to give more meaningful error messages and is thus essential from a practical point of view.

9.2.7 Extra methods required for UNIT and NUMR

In summary, the analysis has shown that equation-oriented simulation can be handled by the existing Unit Operation and Numerical interfaces, with the addition of a small number of extra methods. These are:

- ESO: get names of variables
- SM Unit: get derivatives df/du
- EO unit: get internal variable number for given Global Variable
- EO unit: get internal variable number for given Public Unit Parameter

Note that this also implies changes to the ESO to deal with the variable names.

9.2.8 Mixer-splitter Design Scenario For Steady-state EO Unit and Simulator

We are now able to revise the design scenario used earlier for steady-state, sequential modular simulation for steady-state EO simulation. As before, the purpose of the scenario is to focus the scope of the interface specifications onto what is required to implement a steady-state

mixer-splitter prototype in an EO simulator. It provides the overall functionality required for this prototype, rather than an engineering specification of a mixer-splitter.

For the purposes of this exercise the following assumptions are made:

- The unit relies on the use of a "conventional" physical properties package to compute any physical properties (e.g. enthalpies) that appear in its equations. In a CAPE-OPEN context, this is done via the material object.

- The unit can set the attributes of materials (e.g. T, P etc.) in "batch mode", that is, an attribute can be set without triggering a recalculation of the material's internal state. When the unit has finished setting up a material, it then asks it to recalculate. This is done to make the operation of the unit efficient by avoiding unnecessary calculations.

- The unit does not need to inquire about the availability of a thermodynamic server. It just asks the materials to perform actions and the material is assumed to invoke the thermodynamic server internally, if it needs to. In a production simulator, of course, the unit would need to do this. It is quite possible that there might be no physical property system connected at all. This is just a simplifying assumption for this initial prototype.

- The unit with one or more material objects, all derived from the same material template.

- The mixer-splitter allows the user to specify the split factors used to direct flow to the outlet streams. It also has the ability to specify heat input.

A) Add a unit to the flowsheet

- User selects a mixer-splitter from a palette of unit operations and places it on the flowsheet.

- The simulator asks each unit to initialise itself passing to it a material template. The unit uses the latter to create a material object(s) and interacts with one of them to obtain the number of components in the system. It uses this information to set up an ESO (Equation Set Object) describing the mathematical system of equations that model the unit's operation.

- The user selects or creates the streams and requests the simulator to connect them, one at a time, to the ports of the unit, possibly by using a GUI to drag the stream to a port on the unit icon or by interpretation of a text input file.

B) Enter unit specific data

- The simulator does not know if a user interface (UI) is available from the unit, so that it must query the unit for one. If a UI is available, it is displayed. If not, the simulator attempts to construct one from the unit's list of Public Unit Parameters.

- When all the user-desired data has been entered, then the unit checks the data for validity and consistency. Note that an EO unit does not necessarily require all the possible data to be specified, unlike a SM unit.

C) Define unit report

- User asks the unit which reports it can produce.

- The unit gives a list from which the user selects.

D) Run a simulation

I) Building the mathematical problem

- The simulator asks the ESO of each unit in the flowsheet for the numbers of equations and variables in it and uses them to construct the global vectors of equations and variables for the entire flowsheet. The simulator also asks the unit ESO for the number of non-zero partial derivatives, the sparsity structure of the partial derivative matrix (i.e. the row and column numbers of each element) and the possibility of computing the exact value of each element.

- The simulator sets up the equations that describe the connectivity of the units in the flowsheet and the corresponding rows of the partial derivative matrix. To do this, the simulator asks the unit to identify which of its variables are associated with each of the connected streams.

- The simulator sets up the specification equations and the corresponding rows of the partial derivative matrix. To do this, the simulator asks the unit to identify which of its variables are fixed and at what values.

II) Analysing and validating the mathematical problem

- The simulator checks that the mathematical problem assembled at step DI is well-posed (e.g. square, structurally non-singular, etc.). If not, the simulator issues some form of diagnostic information to the user.

III) Solving the mathematical problem

- The simulator creates an ESO (Equation Set Object) which incorporates all the information on the mathematical problem assembled at step DI.

- The simulator calls an NLASystemFactory, passing to it the above ESO, to create an NLASystem. The NLASystem is the combination of a solver and the specific mathematical problem.

- The simulator invokes the NLASystem to solve the simulation problem.

- During the numerical solution, each of the following interactions may occur in any order and frequency, depending on the design of the solver:

  - The NLASystem asks its ESO for its current variable values. The ESO obtains these by querying each unit ESO in turn for its own variables, and then assembling these into a global vector which it returns to the NLASystem. *Note:* this is likely to be one of the first actions performed by any NLASystem, since it can only obtain initial estimates for the variables from its ESO. Thus these initial estimates must be present in, or be generated by, the units before the solution process commences. (The exact point in the sequence at which these estimates are produced will depend on the implementation of the component).

- The NLASystem provides new variable values to its ESO. The ESO then distributes these to each unit ESO in turn, updating the local values formerly held by the units.

- The NLASystem requests the current residual values from its ESO. The ESO obtains most of these by requesting them from the ESOs of its constituent units, and assembles the responses into a global vector. It then appends to this vector the values of the residuals for the connectivity and specification equations, before returning the global vector to the NLASystem.

- The NLASystem requests all the current partial derivative values that can be computed by each ESO. Again these are obtained from the individual units and assembled into a global vector, with information for the connectivity and specification equations appended to it. The ESO then returns this global vector to the NLASystem. (It is assumed that the NLASystem will approximate those partial derivatives that cannot be computed by the units).

E) Report results

- The simulator asks the unit to produce its report.

9.2.9 Extension to Handle EO Dynamic Simulation

The scenario for dynamic simulation problems is very similar to that given above, with the following differences:

- At step B, the initial condition of the unit is specified, in addition to the other data that need to be specified for steady-state problems. If the unit has its own UI, then the latter can handle the specification of this information. If not, the simulator could attempt to handle it by presenting to the user a list of the unit's "state" (differential) variables[1] and asking for their initial values.

- At step DIII, the simulator generally needs to solve at least two distinct mathematical problems:

    - First, the DAE system needs to be initialised. To do this, the simulator starts by forming an extended ESO comprising the ESO constructed at step DI, plus the equations corresponding to the initial condition specifications (cf. step B). It then calls an NLASystemFactory passing to it this extended ESO and asking it to create a NLASystem. It then invokes the latter to solve the initialisation problem. (In this problem, the time derivatives of the differential variables are treated as independent unknowns; hence the extended ESO is square).

    - Assuming the initialisation is completed successfully, the integrator has to proceed with the integration of the dynamic equations over time. To do this, it calls a DAESystemFactory passing to it the ESO constructed at step DIII to create a DAESystem. It then invokes the latter to carry out the integration.

[1] These could be identified using data provided by the unit's ESO.

The interactions of the two solvers mentioned above with their corresponding ESOs are essentially identical to those listed in step DIII above, ultimately involving interactions with the ESOs of the individual units in the flowsheet.

- Reporting of dynamic results: If the reporting is to be done by the units (as in the steady-state case), then we must make sure that the simulator calls the individual unit reporting methods at the frequencies required by them *during* the solution method - not just at the very end – which may require changes at steps C and E.