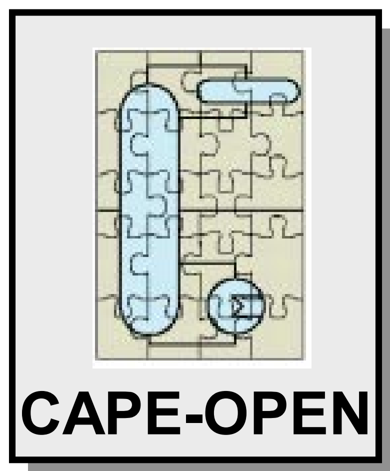
# OPEN INTERFACE SPECIFICATION
# NUMERICAL SOLVERS




*CO-NUMR-EL-03 Version 1*

# 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 seller 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.

# CAPE-OPEN Archival Information

| | |
|---|---|
| **Reference** | CO-NUMR-EL-03 Version 1.08 |
| Coordinated by | ELF |
| Date | 30 June 1999 |
| Number of Pages | 107 |
| Version | Version 1.08 |
| Filename | CAPE-OPEN Component Solver.doc |
| **Developmental Editor(s)** | Costas Pantelides, Imperial College<br>Ben Keeping, Imperial College<br>Jacky Bernier, ELF<br>Christian Gautreau, ELF |
| **Copy Editor(s)** | Michel Pons, ELF |
| **Proofreading Editor(s)** | |
| **Contributor(s)** | Wolfgang Marquardt, RWTH-LPT<br>Jean-Marc Le Lann, INPT<br>Jean-Pierre Belaud, INPT<br>Alain Sargousse, INPT<br>Renée Esposito, ELF<br>Daniel Leineweber, BAYER |

# Summary

This document, which was produced by the Numerical work package, describes the Interface Specifications for the Solver component of the CAPE-OPEN Interface System. This document is written from the point of view of the software developer who will be interested in developing CAPE-OPEN compliant software. Thus, the emphasis in the main body of the document is on the precise definitions of the concepts and interfaces. In particular, it includes detailed definitions (as well as prototype code) for the interfaces relating to the solution of linear and nonlinear algebraic equations, and the core functionality required for solution of differential-algebraic systems (i.e. dynamic problems**).**

The document starts with a textual description of the requirements identified for an open solver component. This is then expressed in Unified Modelling Language and developed into a specification of the interfaces necessary for a CAPE-OPEN solver component to plug into a compliant flowsheet simulator. These specifications are provided in CORBA IDL.

# CAPE-OPEN Document Roadmap

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

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

## Acknowledgements

The authors of this document wish to acknowledge the contribution of the various people whose names appear in the archival section. Without their help, thoughts and technical expertise we would not have been able to produce this document. We would also like to thank the companies involved for supporting this project and providing the significant resources needed to carry out the work.

# Contents


1 INTRODUCTION .......... 1

2 REQUIREMENTS .......... 2

**2.1 User requirements for an Open Solver Component .......... 2**
2.1.1 Setting the Scene .......... 2
2.1.2 Architecture .......... 3
2.1.3 CO Objects Present in a Compliant Simulator .......... 5
2.1.4 Models .......... 6
2.1.5 Mathematical Problems .......... 6
2.1.6 Solvers .......... 6
2.1.7 Solver Manager .......... 7
2.1.8 The Equation Set Object .......... 7
2.1.9 Description of Discontinuous Processes .......... 8
2.1.9.1 Origins of discontinuities in physical descriptions .......... 8
2.1.9.2 Mathematical descriptions of physical discontinuities .......... 8
2.1.9.3 State-Transition Networks .......... 10
2.1.9.4 State-Transition Networks in Models .......... 11
2.1.10 Events and EventInfos .......... 12
2.1.10.1 Events .......... 13
2.1.10.2 EventInfos .......... 15
2.1.11 Overview of Typical Usage .......... 15
2.1.11.1 Modular-Based Dynamic Simulation .......... 15
2.1.11.2 Equation-Based Dynamic Simulation .......... 16
2.1.12 Linear Algebra .......... 17
2.1.12.1 The *FullMatrix* Subtype .......... 17
2.1.12.2 The *UnstructuredMatrix* Subtype .......... 18
2.1.12.3 The *BandedMatrix* Subtype .......... 19
2.1.13 Desirable Characteristics for the CO Interface .......... 20

**2.2 Use Cases .......... 21**
2.2.1 Use Cases Categories .......... 21
2.2.2 Actors .......... 22
2.2.3 Use Cases .......... 23
2.2.3.1 Solver Selection, Instantiation and Configuration .......... 23
2.2.3.1.1 Select Numerical Code (ref. UC-41-001) .......... 24
2.2.3.1.2 Unit Selects Numerical Code (ref. UC-41-002) .......... 25
2.2.3.1.3 Configure Numerical Code (ref. UC-41-003) .......... 26
2.2.3.2 Solver Initialisation .......... 27
2.2.3.2.1 Unit Defines Linear Equations to be Solved (ref. UC-41-004) .......... 28
2.2.3.2.2 Unit Defines Nonlinear Equations to be Solved (ref. UC-41-005) .......... 29
2.2.3.2.3 Unit Defines DAEs to be Solved (ref. UC-41-006) .......... 30
2.2.3.2.4 Identify Global System to Be Solved (ref. UC-41-007) .......... 31
2.2.3.2.5 Eliminate Degrees of Freedom (ref. UC-41-008) .......... 32
2.2.3.3 Solver Execution .......... 33
2.2.3.3.1 Unit Requests Solution of Unit Equations (ref. UC-41-009) .......... 34
2.2.3.3.2 Converge Nonlinear Problem (ref. UC-41-010) .......... 35
2.2.3.3.3 Advance DAE Solution (ref. UC-41-011) .......... 36
2.2.3.4 More Complex Use Cases .......... 37
2.2.3.4.1 Perform EO Steady-State Simulation (ref. UC-41-012) .......... 38
2.2.3.4.2 Perform SM Steady-State Simulation (ref. UC-41-013) .......... 39
2.2.3.4.3 Perform EO Dynamic Simulation (ref. UC-41-014) .......... 40
2.2.3.4.4 Perform SM Dynamic Simulation (ref. UC-41-015) .......... 42

3 ANALYSIS ........ 43

3.1 Overview ........ 43

3.2 Component Diagram ........ 44

Figure 3-1 : Package dependencies between Components ........ 44

Class Diagrams ........ 45
3.2.1 Class Diagram of the Unit Package ........ 46
3.2.2 Class Diagram of the Model Package ........ 47
3.2.3 Class Diagram of the ESO Package ........ 49
3.2.4 Class Diagram of the Solver Package ........ 50

3.3 Sequence Diagram ........ 51

3.4 Collaboration diagram ........ 52

3.5 Interface Diagrams ........ 53
3.5.1 Model Interface Diagram ........ 54
3.5.2 ESO Interface Diagram ........ 55
3.5.3 Solver Interface Diagram ........ 56

3.6 Interface Descriptions ........ 57
3.6.1 Model Component ........ 59
3.6.1.1 Model Manager : ICapeNumericModelManager ........ 60
3.6.1.2 Simulation Model: ICapeNumericModel ........ 61
3.6.1.3 Continuous Model: ICapeNumericContinuousModel ........ 73
3.6.1.4 Hierarchical Model: ICapeNumericHierarchicalModel ........ 73
3.6.1.5 Aggregate Model: ICapeNumericAggregateModel ........ 74
3.6.1.6 State Transition Network: ICapeNumericSTN ........ 76
3.6.1.7 Event : ICapeNumericEvent ........ 85
3.6.1.8 Basic Event : ICapeNumericBasicEvent ........ 88
3.6.1.9 Composite Event : ICapeNumericCompositeEvent ........ 91
3.6.1.10 Binary Event : ICapeNumericBinaryEvent ........ 93
3.6.1.11 Unary Event : ICapeNumericUnaryEvent ........ 94
3.6.1.12 Event Info : ICapeNumericEventInfo ........ 94
3.6.1.13 External Event Info : ICapeNumericExternalEventInfo ........ 98
3.6.1.14 Internal Event Info : ICapeNumericInternalEventInfo ........ 98
3.6.2 ESO Component ........ 100
3.6.2.1 Internal types used by this component ........ 101
3.6.2.2 Matrix interface : ICapeNumericMatrix ........ 101
3.6.2.2.1 ICapeNumericFullMatrix ........ 108
3.6.2.2.2 CapeNumericUnstructuredMatrix ........ 108
3.6.2.2.3 CapeNumericBandedMatrix ........ 109
3.6.2.3 Equation Set Object Manager interface : ICapeNumericESOManager ........ 110
3.6.2.4 Equation Set Object (ESO) interface : ICapeNumericESO ........ 111
3.6.2.5 Linear Analysis ESO interface : ICapeNumericLAESO ........ 128
3.6.2.6 Non Linear Analysis ESO interface : ICapeNumericNLAESO ........ 133
3.6.2.7 Differential Analysis ESO interface : ICapeNumericDAESO ........ 134
3.6.2.8 Global ESO interface : ICapeNumericGlobalESO ........ 143
3.6.2.9 Global LAESO interface : ICapeNumericGlobalLAESO ........ 145
3.6.2.10 Global NLAESO interface : ICapeNumericGlobalNLAESO ........ 145
3.6.2.11 Global DAESO interface : ICapeNumericGlobalDAESO ........ 145
3.6.3 Solver Component ........ 146
3.6.3.1 Solver Manager interface : ICapeNumericSolverManager ........ 147
3.6.3.2 Numeric Solver interface : ICapeNumericSolver ........ 148
3.6.3.3 Numeric LA Solver interface : ICapeNumericLASolver ........ 156
3.6.3.4 Numeric NLA Solver interface : ICapeNumericNLASolver ........ 157

3.6.3.5 Numeric DAE Solver interface : ICapeNumericDAESolver........................................163

# 4 INTERFACE SPECIFICATIONS ............................................................... 169

**4.1 CORBA IDL ...................................................................................................................169**
4.1.1 Common definitions ...................................................................................................169
4.1.1.1 Utility Definitions ...............................................................................................169
4.1.1.2 Utility Component ..............................................................................................170
4.1.2 Model Component .....................................................................................................170
4.1.3 ESO Component ........................................................................................................176
4.1.4 Solver Component .....................................................................................................180

**4.2 COM IDL.....................................................................................................................182**
4.2.1 Common definitions ...................................................................................................182
4.2.2 Model Component .....................................................................................................182
4.2.3 ESO Component ........................................................................................................182
4.2.4 Solver Component .....................................................................................................182

# 5 NOTES ON ANALYSIS AND INTERFACE SPECIFICATIONS................. 183

**5.1 Differences between CORBA and COM .......................................................................183**

**5.2 Public Parameter...........................................................................................................183**

**5.3 Reporting .....................................................................................................................186**

**5.4 Thermo and Physical Property Packages ....................................................................188**

**VERZEICHNISSEITENREFSEITENREFSEITENREF**SEITENREFSEITENREFSEITENREF
SEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREF
SEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENR
EFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREF**SEITENREF**SEITENREFSEITE
NREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITE
NREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFS
EITENREFSEITENREFSEITENREFSEITENREFSEITENREF**SEITENREFSEITENREFSEITE**
**NREFSEITENREFSEITENREF**SEITENREFSEITENREFSEITENREFSEITENREF**SEITENREFSEIT**
**ENREFSEITENREF**SEITENREFSEITENREFSEITENREF**SEITENREF**SEITENREFSEITENREFSEIT
ENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFS
EITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENRE
FSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITEN
REFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEITENREFSEIT
ENREFSEITENREFSEITENREF**SEITENREFSEITENREF**SEITENREFSEITENREFSEITENR
EFSEITENREFSEITENREFSEITENREF**SEITENREF**SEITENREFSEITENREFSEITENREFSEITE
NREF**SEITENREFSEITENREFSEITENREFSEITENREFSEITENREF**

# 1 Introduction

This document aims at defining CAPE-OPEN standard interfaces for Numerical Solvers. The document starts with a textual description of the requirements identified for an open solver component. This is done by introducing the key concepts on which these interfaces are based. These key concepts are supported by a number of examples drawn from the current usage of simulators.

The main body of this document describes the interfaces. A Unified Modelling Language (UML) description is given. It begins with describing the actors and use cases. Indeed, in the UML process, the first step is to express the user requirements in the form of a Use Case Model. 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. Then the interfaces, their methods and their collaboration process are expressed in a series of diagrams.

The document provides the interfaces necessary for a CAPE-OPEN solver component to plug into a compliant flowsheet simulator. These specifications are provided in CORBA IDL.

The work process involved in the NUMR work package led to the production first of a Conceptual Requirements document outlining what is the typical usage of solvers within a process simulator. From this document, a list of use cases was drawn which then led to a number of Interim Interface Specification Documents which were progressively modified through several reviewing steps. Prototyping was developed concurrently and demonstrations took place with the prototypes issued. The current document is a synthesis of all of these documents and the learning obtained to date from the prototyping exercise.

# 2 Requirements

This chapter is devoted to a description of the usage made of solvers within process simulators. After a short reminder of the general goal of setting Open interfaces between the components of a simulator, a few key concepts are defined that will be central to the interface model developed in the next chapter. Typical usage of solvers within simulators is then textually described before listing the main Use Cases considered in our approach.

## 2.1 User requirements for an Open Solver Component

### 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:

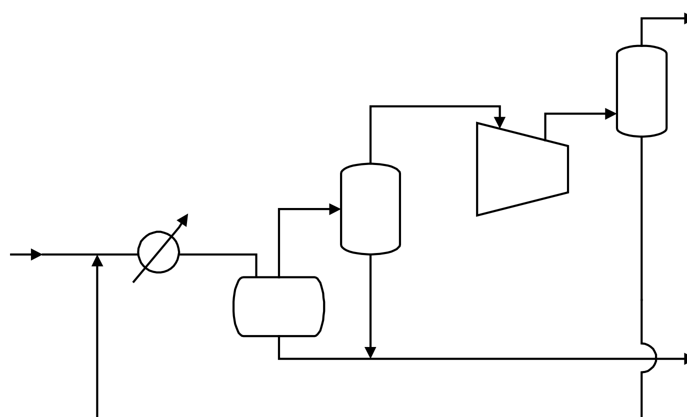


**Figure 2-1 Single stage from an oil and gas separation system**

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 2-1.

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.

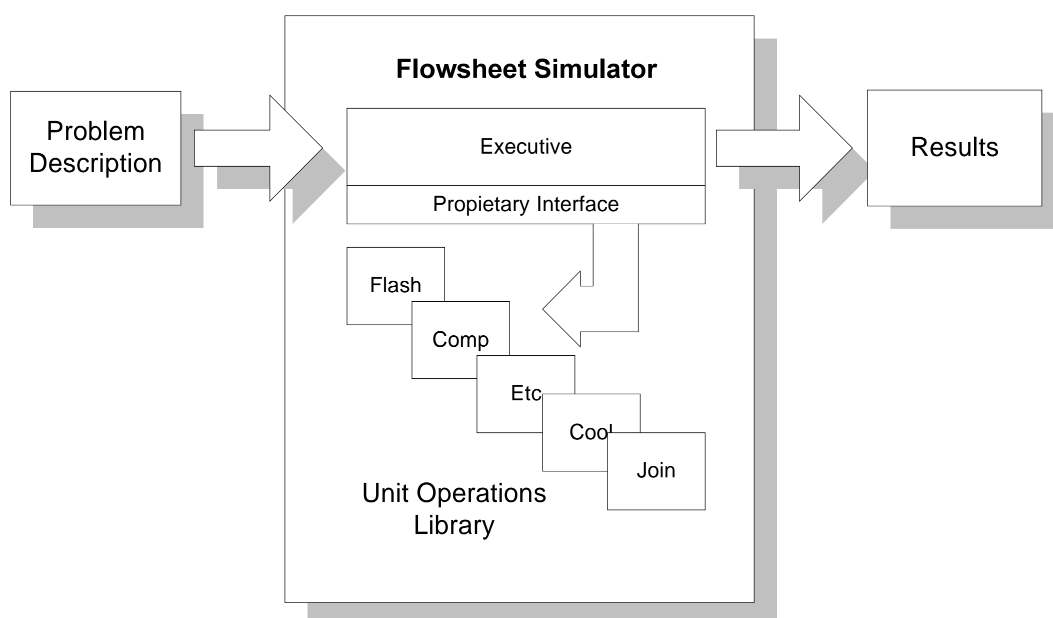



**Figure 2-2 Flowsheet simulator schema**

The CAPE-OPEN (CO) project envisages a new situation whereby a solver (and other simulator software sub-systems, such as thermodynamic or unit packages) can be bought off-the-shelf and plugged directly into any compliant simulator without modification, compiling or linking and will continue to work with subsequent versions of the simulator. All that is required is that the solver and the simulators conform to the CAPE-OPEN Interface System. This System will be defined by the CO project, which is 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 NUMR work package. It starts by describing the requirements of the part of the CAPE-OPEN Interface System dealing with solvers. These requirements are further described in Unified Modelling Language (UML).

### 2.1.2 Architecture

The architecture of the CO system is based on an object-orientated technology, which allows software systems to be constructed from 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. The same has to be true for any solver component which can be thought as a plug, exchanging information with the socket from a UO or from the simulator executive.

For example, these diagrams show some of the ways in which external facilities could communicate with a host simulator through the interface that has been 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 work packages, rather than any physical segregation in the final interface.**

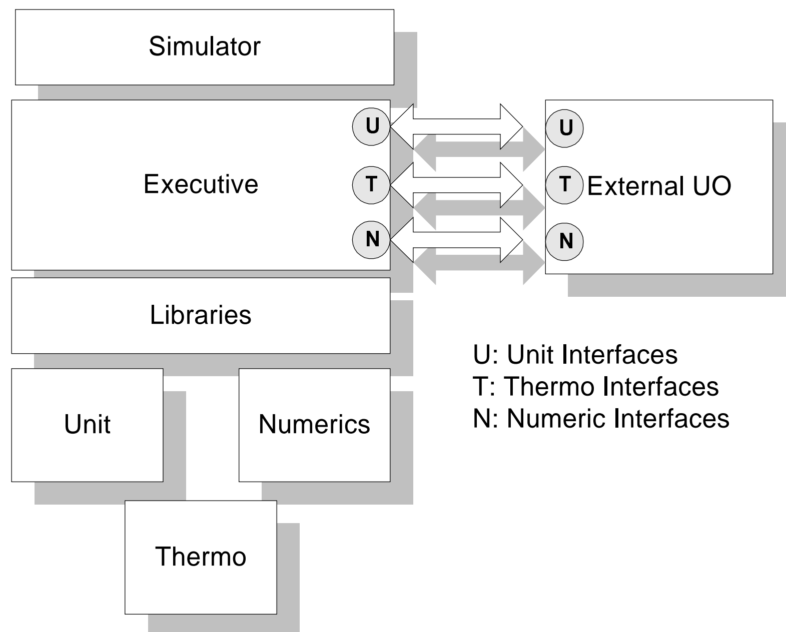



**Figure 2-1 Areas of responsibility of the different work packages (approach 1)**

This shows an external UO using the host simulator's thermodynamic facilities. The UO is solving its own equations and so is unlikely to be sending numerical information to the simulator. This is likely to be the default method of operation in sequential modular simulators.

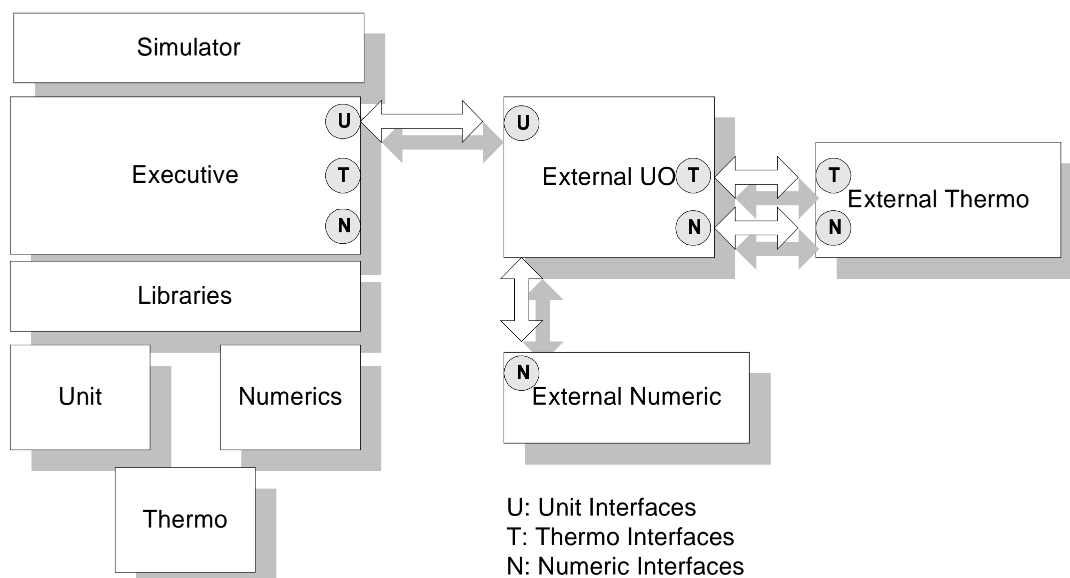



**Figure 2-2 Areas of responsibility of the different work packages (approach 2)**

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. Numerical components may be of two kinds. First sequential modular simulators require specific tools to address graph analysis problems. This is adressed by the SMST component. Then, both types of simulator need solvers of different set of equations modelling the behaviour of individual unit operations or of the

complete process. This is addressed by the SOLVER component. This document is focusing on the latter component.

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 Models

We introduce the *Model* object to embody the general mathematical description of a physical system. The fundamental building block employed for this purpose is a set of continuous equations, described by an *Equation Set Object* (see section 2.1.8).

However, many physical systems also involve discontinuities (see section 2.1.9.1), and this fact must be reflected in their mathematical description. Accordingly, a Model may additionally encompass one or more *State Transition Networks*. These are formal descriptions of discontinuous phenomena (see section 2.1.9.3).

### 2.1.5 Mathematical Problems

We are concerned with the solution of three different types of mathematical problems that are relevant to the operation of flowsheeting packages:

1. The solution of square systems of linear algebraic equations.
2. The solution of square systems of nonlinear algebraic equations.
3. The solution of mixed square systems of ordinary differential and algebraic equations (DAEs) over time or another independent variable.

All of these problems are relevant to both Sequential/Simultaneous Modular-based and Equation-based flowsheeting packages.

The solution of systems involving partial differential and/or integral equations, and of optimisation problems is considered to be outside the scope of the current CAPE-OPEN project.

### 2.1.6 Solvers

We propose to achieve the above aims by introducing three different classes of object, each corresponding to one of the problems listed above. In the rest of this document, we will generically refer to these objects as "*Solvers*":

1. The Linear Algebraic Solver (*LASolver*) object.
2. The Nonlinear Algebraic Solver (*NLASolver*) object.
3. The Differential-Algebraic Equation Solver (*DAESolver*) object.

Each of these contains both the data that characterise the corresponding mathematical problem *and* the numerical algorithms that solve this problem.

### 2.1.7 Solver Manager

In addition to the three types of Solvers, we introduce one "manager" class. This is used to create Solvers using information that defines the mathematical problem to be solved by each such instance.

### 2.1.8 The Equation Set Object

The definition of large sets of nonlinear equations of any kind generally requires a large amount of relatively complex data. This has led us to introduce the concept of an *Equation Set Object* (ESO) as a means of defining this information in a way that can be accessed and used by instances of NLASolvers and DAESolvers. The structure of the ESO is, therefore, central to the interface definitions which are the ultimate goal of this work.

The Equation Set Object is an abstraction representing a square or rectangular set of equations. These are the equations that define the physical behaviour of the process[1] under consideration, and which must be solved within a flowsheeting problem. The interface to this object is intended to serve the needs of the various solver objects by allowing them to obtain information about the size and structure of the system, to adjust the values of variables occurring in it, and to compute the resulting equation residuals and, potentially, other related information (*e.g.* partial derivatives). Hence, this interface requires standardisation as part of CAPE-OPEN. However, the *construction* of such an object will be a proprietary matter for individual vendors of flowsheeting packages and will *not* be standardised as part of CAPE-OPEN.

More specifically, an ESO will support a number of operations including the following:

- Obtain the current values of a specified subset of the variables.
- Alter the values of any specified subset of the variables.
- Compute the residuals of any specified subset of the equations at the current variable values.
- Obtain the partial derivatives of a specified subset of the equations with respect to a specified subset of the variables (at the current variable values of the object).

The information associated with an ESO differs depending on whether the set of equations being described is purely algebraic (as is the case with the NLASolver class mentioned above) or mixed differential and algebraic (as in the case of DAESolver). For this reason, we introduce a hierarchy of ESOs. At present, this hierarchy comprises three classes:

1. Class *AlgebraicESO* defines a linear system of equations.

[1] Here, the term "process" may mean the entire plant being modelled, a plant section or, indeed, a single unit operation or part thereof.

2. Class *DifferentialAlgebraicESO* inherits from class AlgebraicESO and refines it to define a mixed set of differential and algebraic equations.

An underlying assumption throughout this document is that we are often dealing with large, sparse mathematical systems. Hence the exploitation of sparsity is an important consideration.

### 2.1.9 Description of Discontinuous Processes

The ESO is a purely continuous mathematical description; this means that the equations it contains have the *same* form for all possible values of the variables occurring in them.

However, our best understanding of a number of common process phenomena is based on *discontinuous* descriptions. As mentioned earlier, the Model concept used to represent nonlinear algebraic and differential problems contains an ESO of the appropriate type and may also carry additional discontinuous information in the form of one or more State Transition Networks (the States of which are defined through further Models). As we will see, this permits complex hierarchies of discontinuous behaviour to be represented in a natural way.

However, it is worth stating at the outset that it is **not** our intention to require all solvers for these types of problem to include algorithms for handling discontinuous problems. The design of the Model object is such that it is a simple matter for codes which lack such facilities to check whether any STNs are in fact present, and to report a failure if so.

#### 2.1.9.1 Origins of discontinuities in physical descriptions

There are many examples of process phenomena that are commonly described in a discontinuous manner. These include:

- appearance and disappearance of thermodynamic phases;
- transitions of flow regimes from laminar to turbulent, and vice-versa;
- changes in the direction of flow, and their consequences;
- changes in flow due to discontinuities in equipment geometry (e.g. position of overflow pipes);
- equipment breakdown.

Additional discontinuities may arise as a result of discrete control actions and disturbances imposed on the process by external agents. However, here we are primarily concerned with discontinuities in the physical behaviour since it is precisely this behaviour that ESOs are supposed to describe mathematically.

#### 2.1.9.2 Mathematical descriptions of physical discontinuities

The mathematical descriptions of physical discontinuities is itself discontinuous. Early modelling tools described such discontinuities via the use of conditional equations typically defined using `IF/THEN/ELSE` constructs. Each such conditional equation has one of two different forms depending on the value (`TRUE` or `FALSE`) of a logical condition. The latter is itself expressed in terms of the values of the system variables.

As an example, consider the friction factor $f$ for flow in a pipe. This is a different function of the Reynolds number $\mathrm{Re}$ depending on whether the flow is laminar or turbulent. Mathematically, this effect is described by the following conditional equation:

```
IF Re < 2100 THEN
```

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

```
ELSE
```

$$\frac{1}{\sqrt{f}} = -4\log_{10}\left(\frac{1.26}{\mathrm{Re}\sqrt{f}} + \frac{\varepsilon / D}{3.7}\right)$$

Albeit by far the simplest mechanism for specifying discontinuous equations, `IF/THEN/ELSE` equations are not sufficiently general for the description of the range of phenomena occurring in chemical processes. For instance, they are unable to describe:

- Asymmetric discontinuities such as the hysteresis phenomena that occur in the opening and closing of safety relief valves; such valves tend to open at a higher pressure than the one at which they actually close.

- Irreversible discontinuities such as those occurring when equipment breaks down when certain operating limits (e.g. pressure) are reached; in most cases, the breakdown, once it occurs, cannot be reversed even if the operating conditions revert to their normal ranges.

For this reason, our description of discontinuities is based on a more general formalism, called *State-Transition Networks*.

### 2.1.9.3 State-Transition Networks

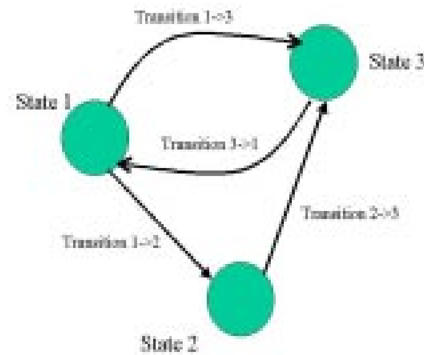



**Figure 2-1 : Example of a State-Transition Network**

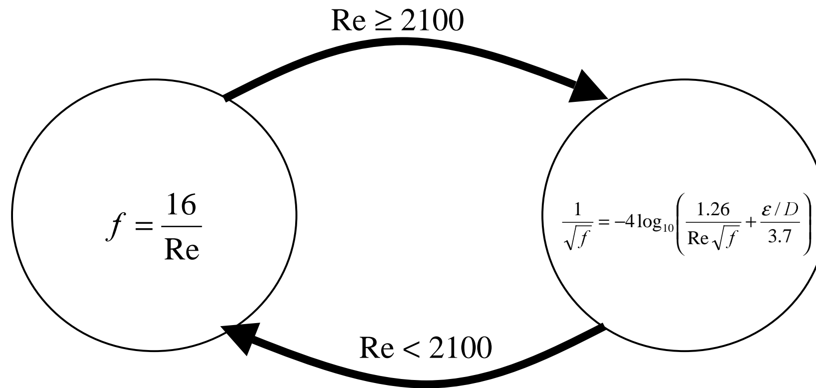
For the purposes of this document, a State-Transition Network (STN) is simply a description of a discontinuous equation or set of equations.

An example of an STN is shown in Figure 2-1. Each STN comprises two types of information, a set of *states*[2] and a set of *transitions* from one state to another.

A state in an STN corresponds to one of the operating regimes of a discontinuous phenomenon. So, for instance, the STN describing the flow regime in a pipe would typically have two states corresponding to the laminar and turbulent regimes respectively.

In fact, `IF/THEN/ELSE` conditional equations (see section 2.1.9.2) are special cases of STNs in which, for any pair of states $s$ and $s'$, both transitions $s \rightarrow s'$ and $s' \rightarrow s$ occur and the logical condition associated with the former transition is the negation of that associated with the latter. Thus, the STN describing the friction factor equation discussed in section 2.1.9.2 is shown in :

[2] The states in a STN are also sometimes called "modes" in order to avoid confusion with the term "states" used in control theory.




More formally, each state *s* in an STN is characterised by:

- A set of equations
- A (possibly empty) set of transitions to other states.

At any particular point in time, *exactly one* state in an STN is designated as being *active*. In physical terms, this implies that the process behaviour satisfies the equations in that state.

Each transition (*s,s'*) in an STN is characterised by:

- A start state, *s*
- An end state, *s'*
- A logical condition.

If, at a certain point in time, a state *s* of the STN is active *and* the logical condition associated with a transition (*s,s'*) becomes `TRUE`, then the transition (*s,s'*) takes place, i.e. state *s* stops being active and state *s'* becomes active.

In the interests of simplicity, all STNs used for the purposes of CAPE-OPEN will satisfy the following assumption:

- All equations and logical conditions are expressed in terms of (subsets of) the *same* set of variables.

#### 2.1.9.4 State-Transition Networks in Models

Each Model contains exactly one ESO, and zero or more STNs. Thus, the complete set of equations which is applicable at any particular point in time comprises:

- the equations in the top level Model's own ESO;
- the equations in the active state of each of its constituent STNs.

Conversely, the equations associated with each state in an STN are themselves described by a Model.

The relation between STNs and Models is, therefore, recursive. This allows for the nesting of discontinuous equations to an arbitrary number of levels.

An underlying assumption is that a Model's own ESO of and all the ESOs describing the states of all STNs contained within it share the *same* set of variables. Albeit not necessary, this assumption obviates the need for complex mapping mechanisms between different sets of variables; it also happens to be satisfied by most typical cases where descriptions of discontinuous phenomena occur within process models.

### 2.1.10 Events and EventInfos

As has already been mentioned in section 2.1.9.3, each transition in an STN is characterised by a logical condition that determines if and when the transition will take place. Such a transition constitutes an *event*.

Another type of event occurs when a dynamic simulation terminates once it has reached a specified time or when the system fulfils a specified condition.

It is important to provide formal mechanisms for representing such events and, in particular, the logical conditions that define them.

Moreover, in designing an object to represent these conditions, it is important to identify the amount and type of information that must be provided to client software regarding them. Clearly, such decisions depend crucially on the type of usage that is envisaged for these conditions by, for instance, numerical solvers[3].

It is important to understand that the above logical conditions may be quite complex. For instance, a transition within an STN could be triggered by a logical condition of the form:

$$\left[ [x_1^2 + x_2^2 \geq x_3^2] \vee \neg [x_1^2 \leq x_2] \right] \quad \wedge \quad [x_2 \geq x_3]$$

where $x_2$, $x_2$ and $x_3$ are real-valued variables, and the symbols $\vee$, $\wedge$ and $\neg$ denote the `OR`, `AND` and `NOT` logical operators.

Most of the older numerical codes for the simulation and optimisation of processes involving discontinuities required only the *evaluation* of complex logical expressions such as the above for given values of the variables occurring in them. Consequently, a simple interface that would return the value (`TRUE` or `FALSE`) of a specified logical condition at the current values of the ESO's variables would be sufficient in this case.

However, more modern solution methods derive their improved reliability and efficiency from the availability of more information on each logical condition. For example, if the above logical condition were to change value (from `TRUE` to `FALSE`, or vice-versa) at a particular point in time, these methods would need to know *exactly which* of the three logical subexpressions:

---

[3] Already similar decisions have been made implicitly in the design of the basic ESO. In that case, it was deemed appropriate that the ESO should provide numerical values for the residuals of its equations and their partial derivatives, as well as information on the structure of these equations. On the other hand, it was not thought necessary to provide information on the symbolic form of these equations.

$$x_1^2 + x_2^2 \geq x_3^2, \quad x_1^2 \leq x_2, \quad x_2 \geq x_3$$

was the one that changed value, thereby causing the change in the value of the overall logical expression. The solution method would also need to know other information on this particular sub-expression, such as the set of variables that appear in it, and its partial derivatives with respect to these variables.

Our aim is to accommodate the requirements of the modern solution methods without resulting in an excessively complex interface. This is particularly important as simpler methods are still being used by many of the currently available tools. As a compromise, the proposed interface supports arbitrarily complex logical conditions involving any combination of the $\vee$, $\wedge$ and $\neg$ operators while imposing the following restriction:

- All lowest level logical sub-expressions in a logical expression are of the form:

  $x_i < k$, $x_i > k$, $x_i \geq k$ or $x_i \leq k$

where $x_i$ is any one of the variables occurring in the ESO.

This assumption is not actually as restrictive as might first appear. For instance, the logical condition shown above could be expressed as:

$$\big[[x_4 \geq 0] \vee \neg[x_5 \geq 0]\big] \quad \wedge \quad [x_6 \geq 0]$$

where we have introduced three new variables $x_4$, $x_5$ and $x_6$ defined via the three additional equations:

$$x_4 \equiv x_1^2 + x_2^2 - x_3^2, \quad x_5 \equiv -x_1^2 + x_2, \quad x_6 \equiv x_2 - x_3$$

#### 2.1.10.1 Events

The main advantage of introducing the above restriction is that it permits the representation of any logical condition by an object known as an *Event* with a relatively simple interface. The event class has three subclasses deriving from it:

- *BasicEvent*, containing a variable index $i$, an operator $op$ and a constant real value $k$. This represents the lowest level condition $x_i \; op \; k$, where $op$ must be either <, >, ≥ or ≤.
- *CompositeEvent*, containing a boolean operator (AND, OR or NOT) and two subevents (only one of which is meaningful in the case of NOT).
- *IndVarEvent*, containing an independent variable value. This type of event represents the most common type of termination condition when solving DAE problems.

Thus, consider the following examples:

- The BasicEvent

  ```
  (3, ≥, 0)
  ```

denotes the condition $x_3 \geq 0$.

- The CompositeEvent

```
[OR, BasicEvent(1, ≥, 0),

    CompositeEvent(NOT, BasicEvent(2, ≥, 0) , ...)]
```

denotes the condition $(x_1 \geq 0) \vee \neg(x_2 \geq 0)$ (i.e. "$x_1$ non-negative or $x_2$ negative").

The pseudocode which follows illustrates how these sequences can be evaluated by a simple recursive routine. We assume that the current variable values are held in the global vector `x`, and introduce enumerated types with value sets (BASIC, COMPOSITE, INDVAR), (GEQ, NEQ), and (AND, OR, NOT) to represent the values of the Event types, the relational operators and the boolean operators respectively.

```
Boolean function eval(Event* event)

  If event->type==BASIC THEN

    If event->op==GEQ THEN

      Return (x[event->index]>=event->value)

    Else

      Return (x[event->index]<=event->value)

    End

  Else

    Case (event->operator)

      AND: Return eval(event->ev1) AND

                  eval(event->ev2);

      OR:  Return eval(event->ev1) OR

                  eval(event->ev1);

      NOT: Return NOT eval(event->ev1);

    End

  End

End
```

#### 2.1.10.2 EventInfos

The occurrence of events is a very important aspect of (especially dynamic) process simulation. If, during the solution process, a Solver object detects an Event associated with the termination of the dynamic simulation, then it naturally has to return control to its client, with some indication of the value of the independent variable (usually time) at which the Event has occurred. Since there may be *multiple* termination events, the Solver also has to indicate which event(s) have actually triggered the termination.

On the other hand, if a Solver detects that an Event associated with an STN transition has taken place, then it essentially has *two* options: *either* handle the event itself *or* return control to its client so that the latter can take some necessary action. Clearly, for this action to be possible, the Solver has to return some information on the events that have actually taken place. In addition to the information associated with termination events, the Solver needs to identify the STN involved and the new state.

We thus define a new class, *EventInfo*, to carry the information associated with the occurrence of an event. It has two sub-classes *ExternalEventInfo* and *InternalEventInfo* deriving from it, because the type of information to be conveyed is rather different in the two cases.

Specifically, when an *internal* event occurs, the Solver (either an NLASolver or a DAESolver) *may* return control to the client software reporting the event, in which case the client software will wish to identify the STN concerned, and the new state to which transition is due to occur. At this point the client software would probably set the STN to the new state, reinitialise the Model in order to obtain variable values consistent with the new set of equations which are now active, and make a further call to the Solver. However, advanced solvers which are able to perform all these actions internally, are not required to return control to the calling routine (although they should call their ReportingInterface routine before and after the discontinuity).

However, when an external event occurs, the Solver (which will be a DAESolver in this case) *must* return control to the client signalling that the event has occured, and in this case the information desired by the client will be an indication of which of the externally specified termination conditions has arisen.

### 2.1.11 Overview of Typical Usage

We will now give a broad outline of the mathematical problems addressed by the two main types of flowsheeting package, namely Modular and Equation-based, highlighting the deployment of the various classes of objects that were introduced and their interactions.

We consider dynamic simulation as an example because it incorporates most of the behaviour relevant to the interface.

#### 2.1.11.1 Modular-Based Dynamic Simulation

Performance of a dynamic simulation in a “typical” modular package may be summarised as follows:

1. Set time to zero and guess torn streams.

2. Ask each unit to initialise itself, using the sequence implied by the torn streams. As part of this initialisation, the unit will create a square DifferentialAlgebraicESO describing its own dynamic behaviour. It will then pass a Model containing this DifferentialAlgebraicESO to a SolverManager to create a DAESolver.

3. Check convergence of torn streams, repeating step 2 if necessary.
4. WHILE not finished DO:

   4.1 Predict the inputs $u(t)$ for all streams.

   4.2 For each unit in the sequence, advance solution from $t$ to $t+\delta$ to obtain the computed stream values $y$. This is the main role of each unit's individual DAESolver.

   4.3 If, for any tear stream $k$, the predicted input $u_k$ differs from the computed input $y_k$ by more than a tolerance $\varepsilon$, reduce $\delta$ and repeat from step 4.1.

5. STOP.

#### 2.1.11.2 Equation-Based Dynamic Simulation

The performance of a dynamic simulation in a "typical" equation-based package may be summarised as follows:

1. Form a single square[4] DifferentialAlgebraicESO $\{f\}$ representing the equations and variables gathered from all the units.
2. Given a set of initial conditions $\{g\}$[5], combine these with the original equations $\{f\}$ to form an augmented square AlgebraicESO $\{f\,g\}$[6].
3. Using the AlgebraicESO created at step 2 (enclosed in a Model), employ a SolverManager to create an NLASolver, and solve the initialisation problem.
4. Using the DifferentialAlgebraicESO created at step 1 (enclosed in a Model), employ a SolverManager to create a DAESolver.
5. Using the DAESolver created at step 4, advance the solution of the dynamic system $\{f\}$ from the point computed at the last step, until a termination condition or discontinuity occurs[7].
6. If a discontinuity has occurred:

---

[4] *i.e.* the number of equations equals the number of variables

[5] Here we consider the initial conditions $\{g\}$ to be general equality relations of the form $g(x(0), \dot{x}(0)) = 0$. The number of such initial conditions will *normally* be equal to the number of differential variables *x*. However, for DAE systems of index exceeding unity, the number will be smaller.

[6] This system is square because the time derivatives $\dot{x}$ are treated as separate unknowns to the *x* variables. The entire system of equations {*f, g*} is then considered to be a purely algebraic system for the purposes of the initialisation calculations.

[7] The detection of discontinuities will typically require *additional* information to be provided by the Model.

6.1 Construct a new augmented square AlgebraicESO $\{f\ R\}$ incorporating continuity or other "junction" conditions $R$ representing the relations between the two stages of the simulation before and after the discontinuity.

6.2 By employing a SolverManager, create an NLASolver using the above AlgebraicESO (enclosed in a Model) and solve the reinitialisation problem.

6.3 Go to step 5.

7 STOP.

### 2.1.12 Linear Algebra

As can be seen from the algorithm sketches in sections 2.1.11.1 and 2.1.11.2, the solution of linear systems is *not* normally a *direct* requirement in either type of package. However, such systems will usually arise as sub-problems in the solution of NLASolvers and DAESolvers. Accordingly, the latter will themselves have responsibility for instantiating and using appropriate LASolvers. For this reason, the interface of LASolvers has to be standardised as part of the CAPE-OPEN activity, and this is why these have been considered in section 2.1.6.

More specifically, we will enable NLASolvers and DAESolvers to carry out the linear algebra operations that they require by making available to them an LASolverManager which they can then use to create LASolvers as and when this is necessary.

In order to allow the code which uses these solvers to be written with the greatest possible generality, we have developed a polymorphic approach to matrices. The top level object involved in this approach is the *Matrix*, which has various subtypes derived from it, *FullMatrix, UnstructuredMatrix* and *BandedMatrix*[8]. The *Matrix* object itself has a *GetValues* method which returns an array of real numbers, but the semantics of this array are dependent on the precise type of the matrix.

The other methods provided by the *Matrix* object itself simply provide the type of the matrix and its dimensions, as well as two Boolean values, "Symmetric" and "ByRow". The first of these indicates whether the matrix is understood to be fully symmetric about the leading diagonal, in which case the repeated values will be omitted in the result of *GetValues.* The second specifies whether (in the unsymmetric, structured cases) the ordering of values is by row or by column: in the former, the values in each row are given in turn (in column order), whereas in the latter the values for each column are given in turn (in row order).

Detailed explanations of the semantics of *GetValues* for each subtype follow.

#### *2.1.12.1* The *FullMatrix* Subtype

This subtype contains no further methods. If "Symmetric" is true, GetValues returns only unique values, thus (in this diagram and those that follow, the index appearing in each cell indicates that element's position in the array returned by GetValues).

[8] Other subtypes can be added if a need for them becomes apparent.

| 1 | 2 | 3 |
|---|---|---|
| 2 | 4 | 5 |
| 3 | 5 | 6 |

Otherwise, if "ByRow" is true, the values are ordered by row:

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

Otherwise, they are ordered by column.

#### 2.1.12.2 The *UnstructuredMatrix* Subtype

This interface adds the following method:

- GetStructure() : row and column indices of nonzeroes.

If "Symmetric" is true, only the lower or up triangular entries should be defined by this method.

There is no ordering requirement on the row and column indices.

Thus the following structure:

| X | 0 | 0 | X |
|---|---|---|---|
| X | 0 | X | 0 |
| 0 | 0 | 0 | X |

Could be represented by

Row list:(1,1,2,2,3)

Column list:(1,4,1,3,4)

or

Row list(1,2,3,2,1)

Column list(1,1,4,3,4)

Thus "ByRow" has no meaning for this type: the semantics of GetValues are determined by the ordering of the lists of indices.

#### 2.1.12.3 The *BandedMatrix* Subtype

This matrix type specifies banded matrices, i.e. those where all nonzeroes occur within some bandwidth of the leading diagonal. The following method is added:

GetBandWidth() – returns an integer N s.t. no nonzero occurs more than N rows/columns from the leading diagonal. All values in this band must then be returned by the GetValues method.

For example, if a 3 by 4 matrix has bandwidth 1, and “ByRow” is true, the semantics of GetValue are:

| | | | |
|---|---|---|---|
| 1 | 2 | | |
| 3 | 4 | 5 | |
| | 6 | 7 | 8 |

while if “ByRow” had been false they would be:

| | | | |
|---|---|---|---|
| 1 | 3 | | |
| 2 | 4 | 6 | |
| | 5 | 7 | 8 |

For symmetric matrices, “ByRow” is irrelevant as for the full case, e.g. 5 by 5 matrix with bandwidth 2:

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | | |
| 2 | 4 | 5 | 6 | |
| 3 | 5 | 7 | 8 | 9 |
| | 6 | 8 | 10 | 11 |
| | | 9 | 11 | 12 |

### 2.1.13 Desirable Characteristics for the CO Interface

- 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.
- The interfaces generated by all work packages should be consistent in design.
- The approach to units of measure conversion should be handled consistently across all work packages and with the host simulator.

## 2.2 Use Cases

The next sections formalise the description of the user requirements for solver operations interfacing in the CAPE-OPEN Interface System, described in the previous section. They provide a Unified Modelling Language (UML) 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.

The rest of this section lists and describes the Actors and Use Cases involving Solvers.

### 2.2.1 Use Cases Categories

- **Solvers Use Cases.** Contains all the Use Cases listed in this document.

- **General Purpose Use Cases.** Use Cases that express a software requirement to handle CAPE-OPEN Solver Components. These Use Cases do not have a direct impact on the CAPE-OPEN interfaces, and therefore the requirement does not need to be met 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 Solver 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 Solver 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 Solver is an actor in other CAPE-OPEN Use Cases different from those corresponding to Solvers (i.e. THRM or UNIT Use Cases)

### 2.2.2 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 iterating 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 nonlinear equations.

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

- **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 Flowsheet Unit or a whole process. The optimiser may use an infeasible path method, wherein the optimising and the flowsheet converging are carried on simultaneously.

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

- **Simulator Executive.** That 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.** This is a part of the Simulator Executive that reports on the outcome of the calculation of the flowsheet. It reports on the state of the streams and units involved in the flowsheet. Note: reporting is done in different ways. Some reporting is done directly by the unit operation on a request from the Simulator Executive. In addition, reporting is done by passing some values from the unit to the reporting system which has a generalised report generating capability.

- **Solver Manager.** A subsystem that handles the selection and configuration of solver “factory” components.

- **NLAE Solver.** A solver responsible for converging a system of nonlinear algebraic equations.

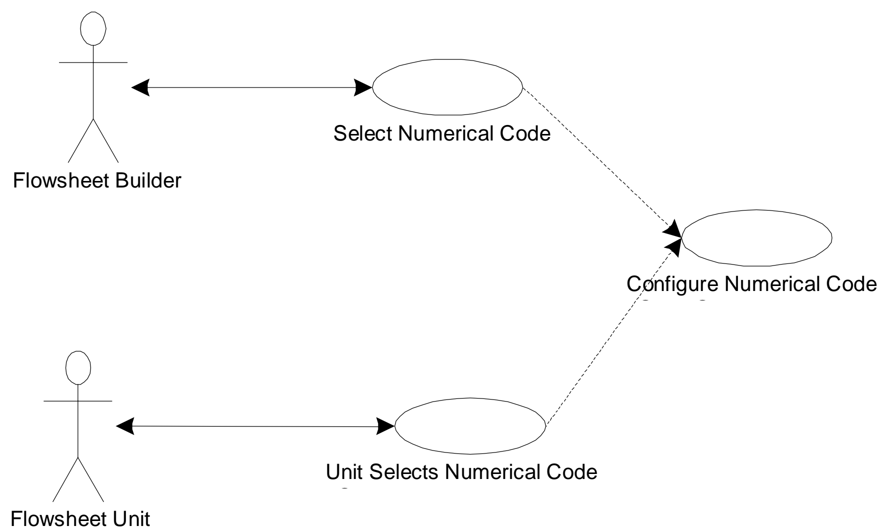
- **DAE Solver.** A solver responsible for advancing the solution of a system of DAEs over the domain of a single independent variable.

### 2.2.3 Use Cases

This subsection lists all the Use Cases that are relevant for the Solver Interfaces.

#### 2.2.3.1 Solver Selection, Instantiation and Configuration

Use Case Diagrams



#### 2.2.3.1.1 Select Numerical Code (ref. UC-41-001)

Actors: <Flowsheet Builder>, <Simulator Executive>, <Solver Manager>

Classification: <Solvers Use Cases>, <General Purpose Use Cases>, <Simulation Context Use Cases>

Status:

Pre-conditions:

- <There must be at least one registered solver of the specified type>
- <A complete flowsheet has been defined>

Flow of events:

*Basic Path:*

The Flowsheet Builder requests the Simulator Executive to carry out a simulation or optimisation.

If no solver for this type of calculation has yet been configured, the Simulator Executive asks the Solver Manager for the list of the numerical codes (*e.g.* DAE solvers) available on the system which are applicable to this task.

The Simulator Executive then displays this list to the Flowsheet Builder who selects the code to be used. The [Configure Numerical Code] Use Case is then applied.

Post-conditions:

- <selection succeeded>

<…>

Exceptions:

- <selection failed>

Subordinate Use Cases:

[Configure Numerical Code (ref. UC-41-003)]

#### 2.2.3.1.2 Unit Selects Numerical Code (ref. UC-41-002)

Actors**:** <Flowsheet Unit>, <Simulator Executive>, <Solver Manager>, <Flowsheet Builder>, <Flowsheet User>

Classification: <Solvers Use Cases>, <General Purpose Use Case>, <Simulation Context Use Case>

Status:

Pre-conditions:

- <There must be at least one registered solver of the specified type>

Flow of events:

*Basic Path:*

The Flowsheet Unit requests from the Solver Manager a list of numerical codes that are appropriate for its purpose. For example, a steady-state flash unit model will typically require the solution of a system of nonlinear algebraic equations; it will therefore request a list of codes for this purpose. On the other hand, a steady-state tubular reactor model may require the solution of a set of differential and algebraic equations over a spatial domain, and would therefore request a list of available DAE integration codes.

The Flowsheet Unit then selects a code from the list itself **or** prompts the Flowsheet Builder/User for a choice. The Solver Manager creates an instance of this code. The [Configure Numerical Code] Use Case is then applied.

Post-conditions:

- <selection succeeded>

<…>

Exceptions:

- <selection failed>

Subordinate Use Cases:

[Configure Numerical Code (ref. UC-41-003)]

#### 2.2.3.1.3 Configure Numerical Code (ref. UC-41-003)

Actors**:** <Solver Manager>

Classification: <Solvers Use Cases>, <Simulation Context Use Cases>

Status:

Pre-conditions:

- <A solver has been selected>

Flow of events:

*Basic Path:*

The Solver Manager asks the Solver for a list of its parameters: each of which will have a name, a type, a default value and a valid range (for real values).

It *may* then provide this list to the user to give him/her the opportunity to override the default values.

Post-conditions:

- <Parameter list obtained>

<…>

Exceptions:

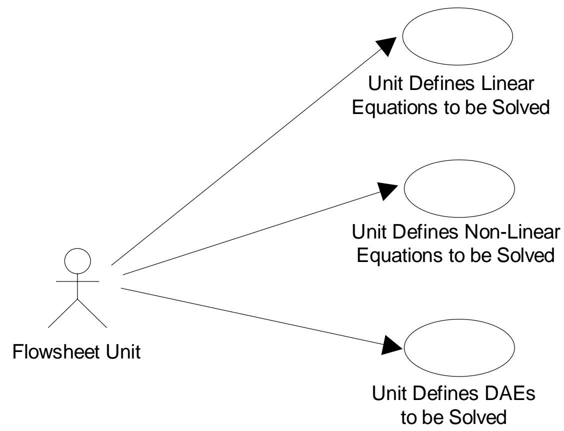
- <Required parameter missing>

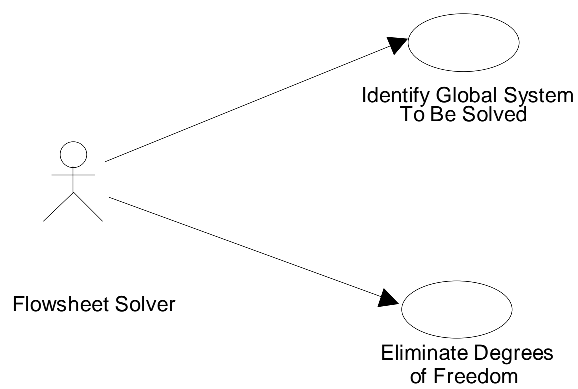
Subordinate Use Cases:

None

#### 2.2.3.2 Solver Initialisation

Use Case Diagrams



#### 2.2.3.2.1 Unit Defines Linear Equations to be Solved (ref. UC-41-004)

Actors**:** <Flowsheet Unit>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- <A Linear Solver has been selected and an instance of it has been created and configured.>

Flow of events:

*Basic Path:*

The Flowsheet Unit provides its Linear Solver with the structure of a matrix $A$ in the equation $Ax = b$ .

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

None

### 2.2.3.2.2 Unit Defines Nonlinear Equations to be Solved (ref. UC-41-005)

Actors**:** <Flowsheet Unit>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

< A Nonlinear solver has been selected and an instance of it created and configured. >

Flow of events:

*Basic Path:*

Summary: The Flowsheet Unit identifies a subset of its equations and variables as a nonlinear problem, and sets up a Nonlinear Solver to handle this problem during execution.

The Flowsheet Unit creates a square set of equations based on $N$ of its equations and $N$ of its variables. It also generates, or otherwise obtains, initial guesses for all unknowns.

Post-conditions:

- ❑ <>

<…>

Exceptions:

- ❑ <>

Subordinate Use Cases:

None

#### 2.2.3.2.3 Unit Defines DAEs to be Solved (ref. UC-41-006)

Actors**:** <Flowsheet Unit>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

< A DAE Solver has been selected and an instance of it created and configured. >

Flow of events:

*Basic Path:*

The Flowsheet Unit identifies a subset of its equations and variables as a differential-algebraic problem, and sets up a DAE Solver to handle this problem during execution. It also generates or otherwise obtains initial guesses for all unknowns.

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

None

### 2.2.3.2.4 Identify Global System to Be Solved (ref. UC-41-007)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

The Flowsheet Solver identifies the global lists of variables and equations. It normally does this by concatenating the sets of equations and variables in the units of the flowsheet, and adding appropriate connectivity information. At this stage, it may also request the Flowsheet Units to provide suitable initial guesses for their variables.

Note: the resulting global set of equations is usually rectangular, involving more variables than equations.

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

None

#### 2.2.3.2.5 Eliminate Degrees of Freedom (ref. UC-41-008)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

The Flowsheet Solver fixes the values of variables which the Flowsheet Builder wishes to be regarded as fixed/known for the present calculation. Thereafter these variables are no longer present in the global variable list. The global variable list should be the same length as the global equation list should be square when this process is complete.

Post-conditions:

- <>

<…>

Exceptions:

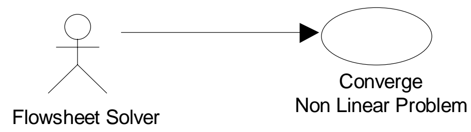
- <Discover inconsistencies in the degrees of freedom specification.>

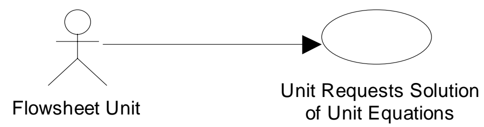
Subordinate Use Cases:

None

### 2.2.3.3 Solver Execution

Use Case Diagrams



#### 2.2.3.3.1 Unit Requests Solution of Unit Equations (ref. UC-41-009)

Actors: <Flowsheet Unit>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < An appropriate solver has been selected, instantiated, configured and initialised.>

Flow of events:

*Basic Path:*

The Flowsheet Unit requests that the numerical method solves the unit equations.

Post-conditions:

- <Unit equations solved>

<…>

Exceptions:

- < Various types of numerical failure may occur. >

Subordinate Use Cases:

None

#### 2.2.3.3.2 Converge Nonlinear Problem (ref. UC-41-010)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < The global equation and variables sets have been constructed. >
- < An appropriate solver has been selected, instantiated, configured and initialised.>

Flow of events:

*Basic Path:*

The Flowsheet Solver requests that the numerical method solves the flowsheet equations**.**

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

None

#### 2.2.3.3.3 Advance DAE Solution (ref. UC-41-011)

Actors: <DAE Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

The DAE Solver carries out steps in the independent variable. It interacts with the global differential-algebraic equation and variable sets as follows :

1) it changes the variable values

2) it requests residual values corresponding to the latest variable values it has supplied

3) it checks for any conditional equations changing their branch.

4) solution must advance exactly to the point where the termination condition is satisfied or a discontinuity occurs (whichever is first).

The termination condition will be provided by the Simulator Executive. It will consist either of an explicit target value of the independent variable, or a condition on a particular variable value.

A discontinuity will result when a conditional equation has changed branch.

The numerical method will involve an iterative procedure. This is likely to make use of a Sparse LAE Solver. It could instead use a NLAE Solver.

Note : this use case is written only for EO simulators.

Post-conditions:

- <>

<…>

Exceptions:

- <>

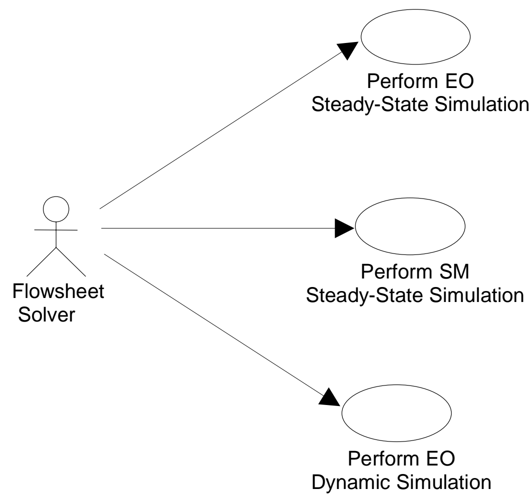
Subordinate Use Cases:

None

### 2.2.3.4 More Complex Use Cases

Note : These use cases have been broken into smaller ones. They should not imply any new use case.

Use Case Diagrams



#### 2.2.3.4.1 Perform EO Steady-State Simulation (ref. UC-41-012)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

The Flowsheet Solver applies the [Identify Global System To Be Solved] Use Case.

If there are any differential variables in the global system, the Flowsheet Solver adds equations setting to zero the derivative of each with respect to the independent variable (time).

The Flowsheet Solver applies the [Eliminate Degrees of Freedom] use case to generate a Nonlinear Problem from the global system.

The Flowsheet Solver then applies the [Converge Nonlinear Problem] use case.

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

[Identify Global System to Be Solved (ref. UC-41-007)]

[Eliminate Degrees of Freedom (ref. UC-41-008)]

[Converge Nonlinear Problem (ref. UC-41-010)]

### 2.2.3.4.2 Perform SM Steady-State Simulation (ref. UC-41-013)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

**[To Be Completed]**

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

[]

#### 2.2.3.4.3 Perform EO Dynamic Simulation (ref. UC-41-014)

Actors**:** <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

< The Flowsheet Builder/User has provided the following information to the Simulator Executive:

1) initial conditions

2) time varying information (*e.g.* scheduled changes in input values)

3) a termination condition>

Flow of events:

*Basic Path:*

The Flowsheet Solver applies the [Identify Global System To Be Solved] use case to construct the system of equations $\{f\}$ which represents the dynamic system, *i.e.* those equations which apply at all values of the independent variable.

The Flowsheet Solver merges this system $\{f\}$ with the initial conditions $\{g\}$ provided by the Flowsheet Builder to create an augmented global system $\{f\ g\}$ for the initialisation.

The Flowsheet Solver applies the [Eliminate Degrees of Freedom] use case to generate a Nonlinear Problem from the augmented global system.

The Flowsheet Solver then applies the [Converge Nonlinear Problem] use case.

The Simulator Executive then enters a loop in which :

1) It instructs the DAE Solver to apply the [Advance DAE Solution] use case to advance to the next discontinuity or to the end of the simulation.

2) If a discontinuity is encountered, it asks the Flowsheet Solver to compute the state of the system immediately after the discontinuity ("reinitialise"). The Flowsheet Solver may do this by :

- Merging the dynamic global system $\{f\}$ with continuity conditions $\{h\}$ (which usually equate the differential variables in their system to their values immediately before the discontinuity) to represent the reinitialisation problem

    - Applying the NonLinear Solver to this problem.

The loop terminates when an error occurs or the termination condition is satisfied.

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

[Identify Global System to Be Solved  (ref. UC-41-007)]

[Advance DAE Solution  (ref. UC-41-011)]

[Eliminate Degrees of Freedom  (ref. UC-41-008)]

[Converge Nonlinear Problem (ref. UC-41-010)]

#### 2.2.3.4.4 Perform SM Dynamic Simulation (ref. UC-41-015)

Actors: <Flowsheet Solver>

Classification: <Solvers Use Cases>

Status:

Pre-conditions:

- < >

Flow of events:

*Basic Path:*

**[To Be Completed]**

Post-conditions:

- <>

<…>

Exceptions:

- <>

Subordinate Use Cases:

[]

# 3 Analysis

## 3.1 Overview

We have defined three separate components in this specification. In particular we have introduced a separation between **the model** (Model Component), **the sets of equations** (ESO Component) and **the solver** itself (Solver Component).

1. The Model describes the physical problem and models the behaviour of a unit or a complete flowsheet, including the physical discontinuities using states and transitions.

2. The ESOs are sets of equations that describes mathematically the continuous part of a particular model or subparts of that model.

3. The Solver itself is responsible of driving the resolution of the problem using all the information defined in the model.

   Inside the Solver Component we have defined three main types of solvers which can be further refined in more specific subtypes :

   - The Linear Algebraic Solver (LASolver)

   - The Nonlinear Algebraic Solver (NLASolver)

   - The Differential Algebraic Equation Solver (DAESolver)

   Each solver type can of course use some of the other types inside that component in order to solve some sub-problems. For example a typical use would be a DAESolver instance creating an instance of a NLASolver for solving some part of the problem or an NLASolver instance using an LASolver.

## 3.2 Component Diagram

In this diagram we have tried to represent the various dependencies between the different components used during a simulation. In our case the UO or the Executive Simulator are clients of both the Model and the Solver, which in turn are making use of the ESO as a common resource.

The interfaces needed between these three components in order for them to communicate and exchange their information are described further in the chapter "Interface description".

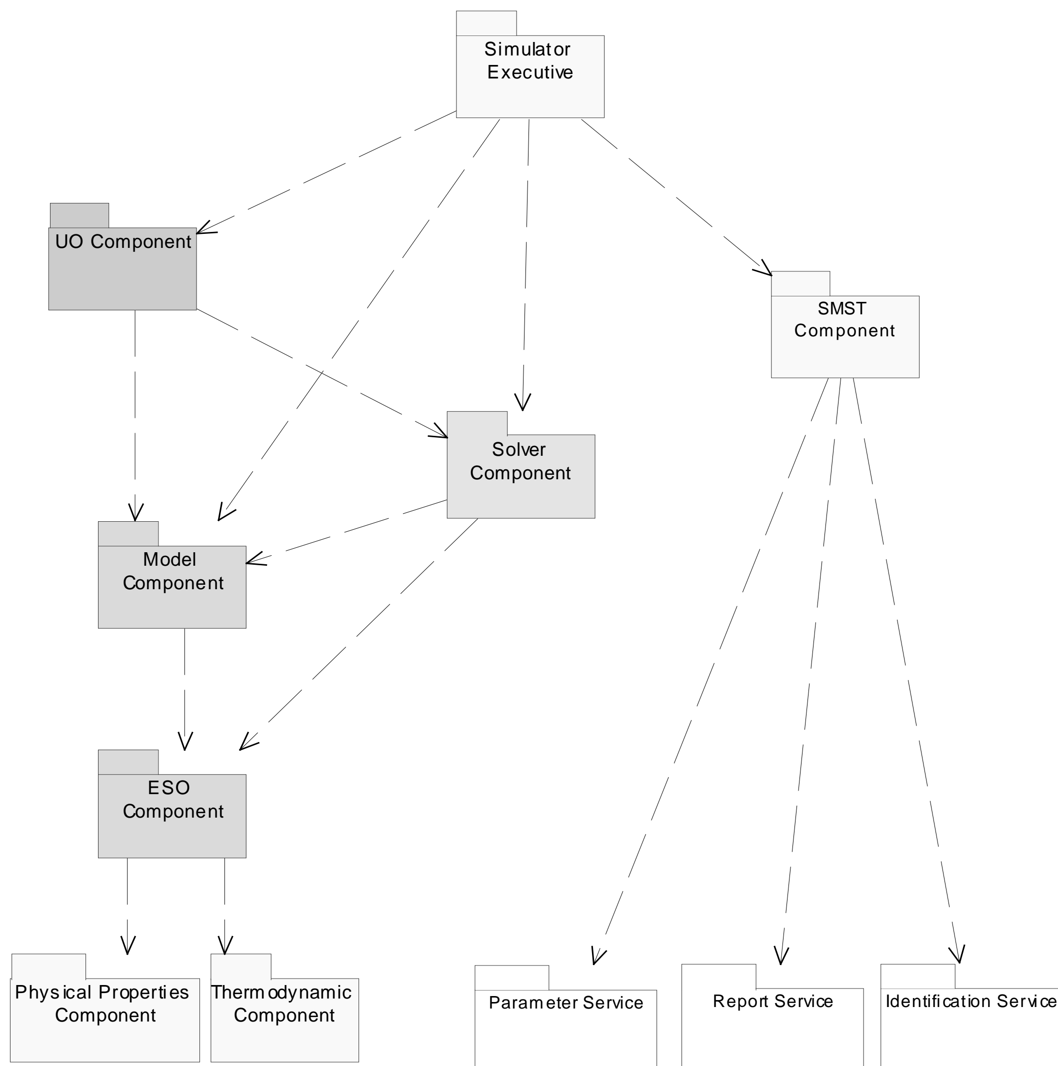



**Figure 3-1 : Package dependencies between Components**

## Class Diagrams

According to our package decomposition, we have defined three class diagrams, one for each package. In these class diagrams, you will also find some specific data types or classes that are implementation dependent and that should not be part of the interface diagram itself. They are there only as an example to promote a better understanding of each diagram.

On the contrary there is some other types, or classes, that are common to different packages (some of them could even be shared throughout the whole CapeOpen project).

Among them we have defined the following classes or data types :

1. The class **Matrix,** which can be further refined if needed into subclasses like FullMatrix, SparseMatrix, BandedMatrix, etc.

2. The class **PhysicalVariable** which describes the different variables of the Unit Operation or of the Simulator Executive. These variables are used by a Model.

3. The class **Event** with its different subclasses (BasicEvent, CompositeEvent, BinaryEvent, UnaryEvent).

4. The type **PublicParameter** which holds information on any parameter in general.

5. The type **NumericVariable** is a type that holds the different values for each variable.

All the classes will be described with the interface description of the component where they are used, but the types will be defined in a separate chapter because they can be used in different interfaces.

### 3.2.1 Class Diagram of the Unit Package

The Unit Operation is not part of this specification, but for a better understanding of the relationships between the UO and the other components, we have included here a simplified representation of this class diagram.

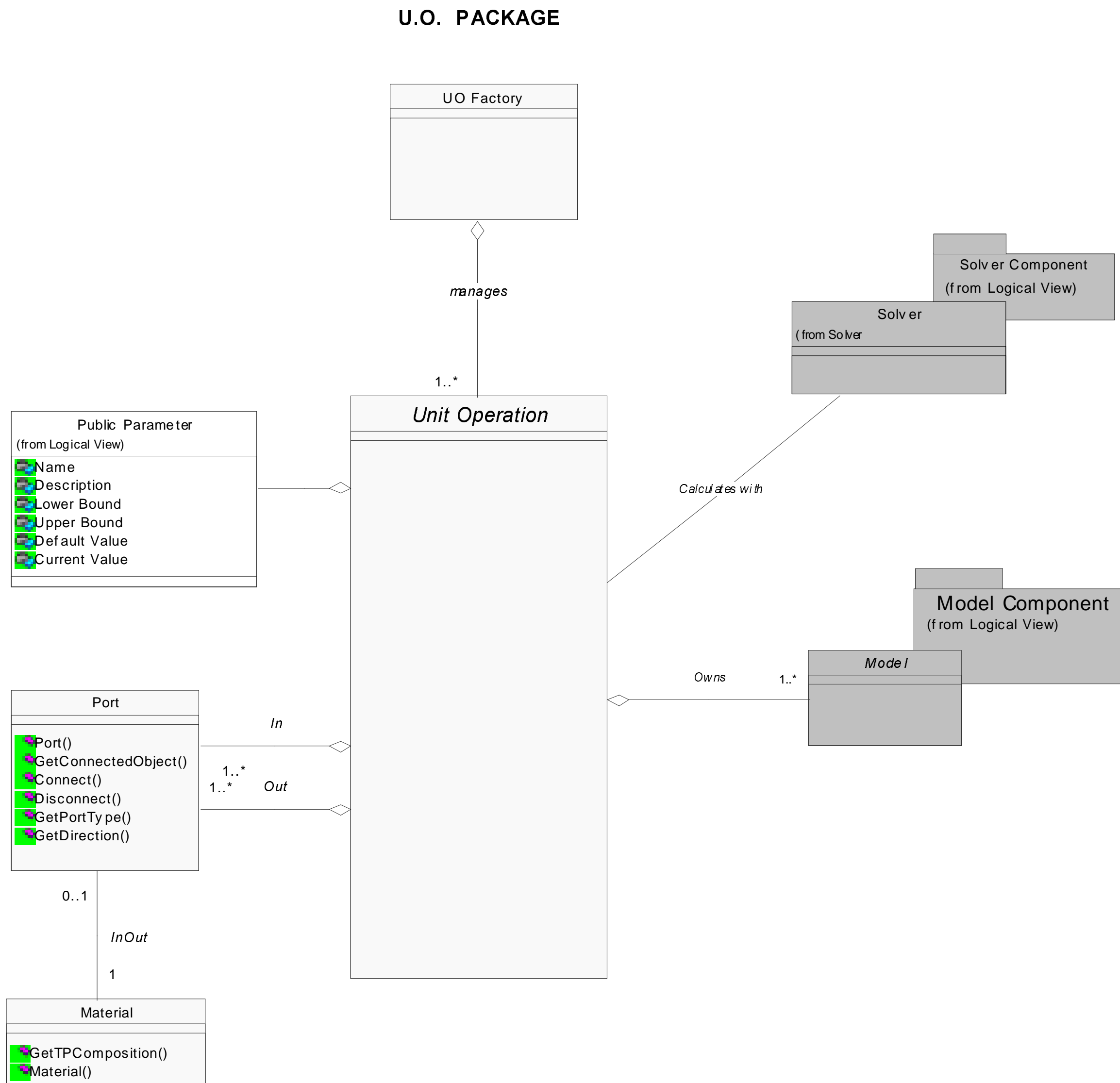



**Figure 3-1 : Simplified diagram of the Unit Operation**

### 3.2.2 Class Diagram of the Model Package

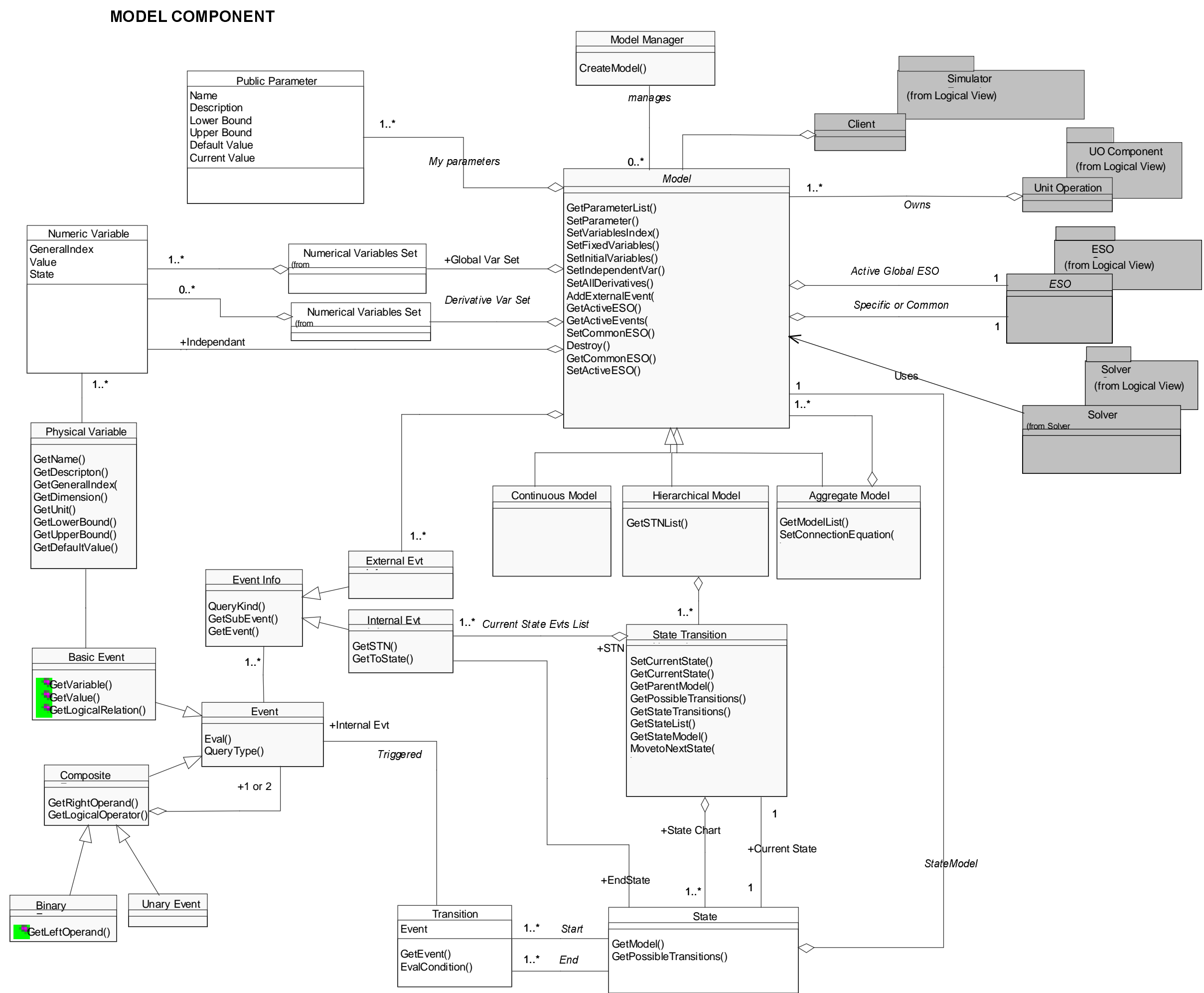



**Figure 3-1 : Class diagram of the Model package**

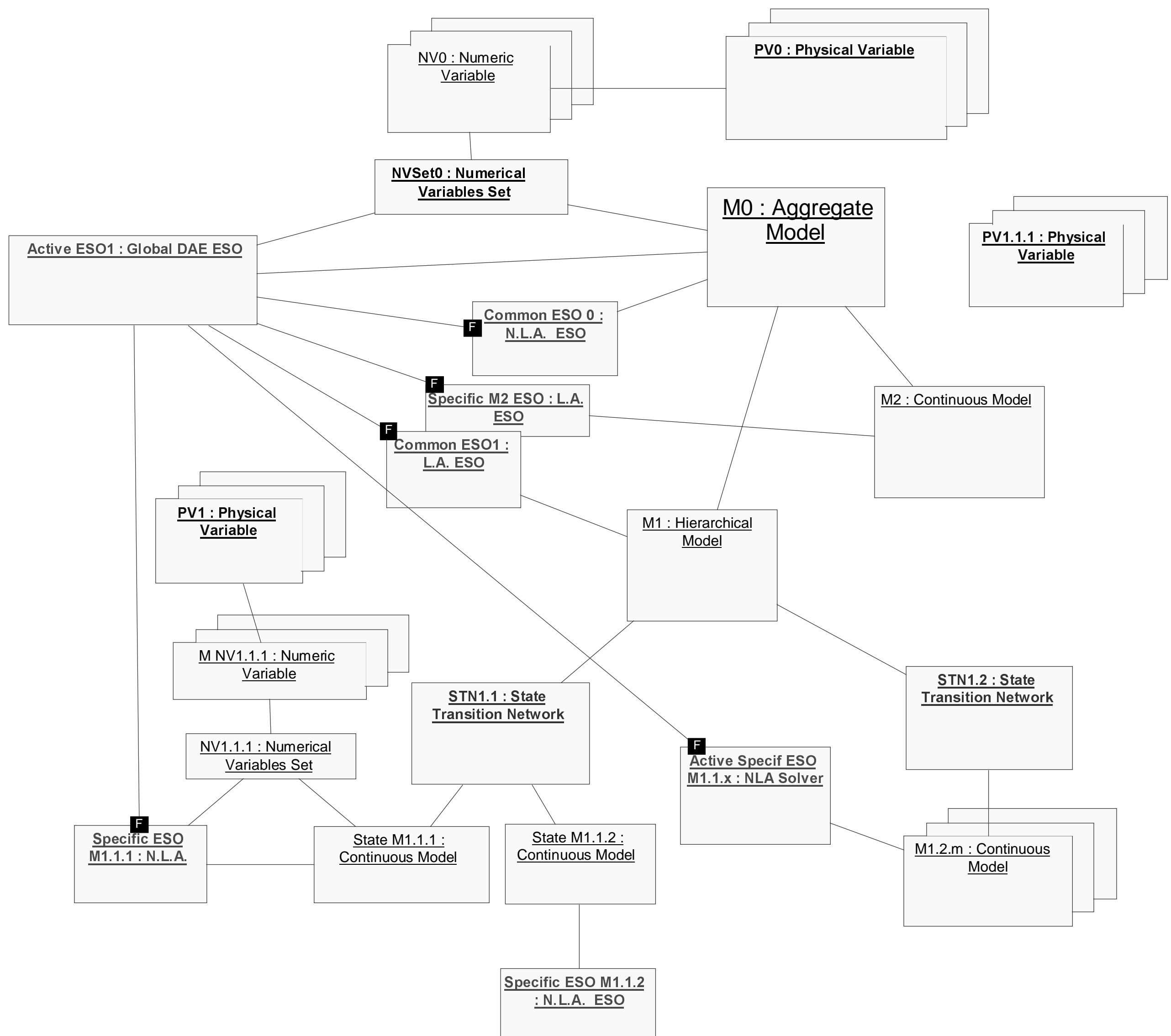



**Figure 3-2 : Sample of a Model instanciation**

## 3.2.3 Class Diagram of the ESO Package

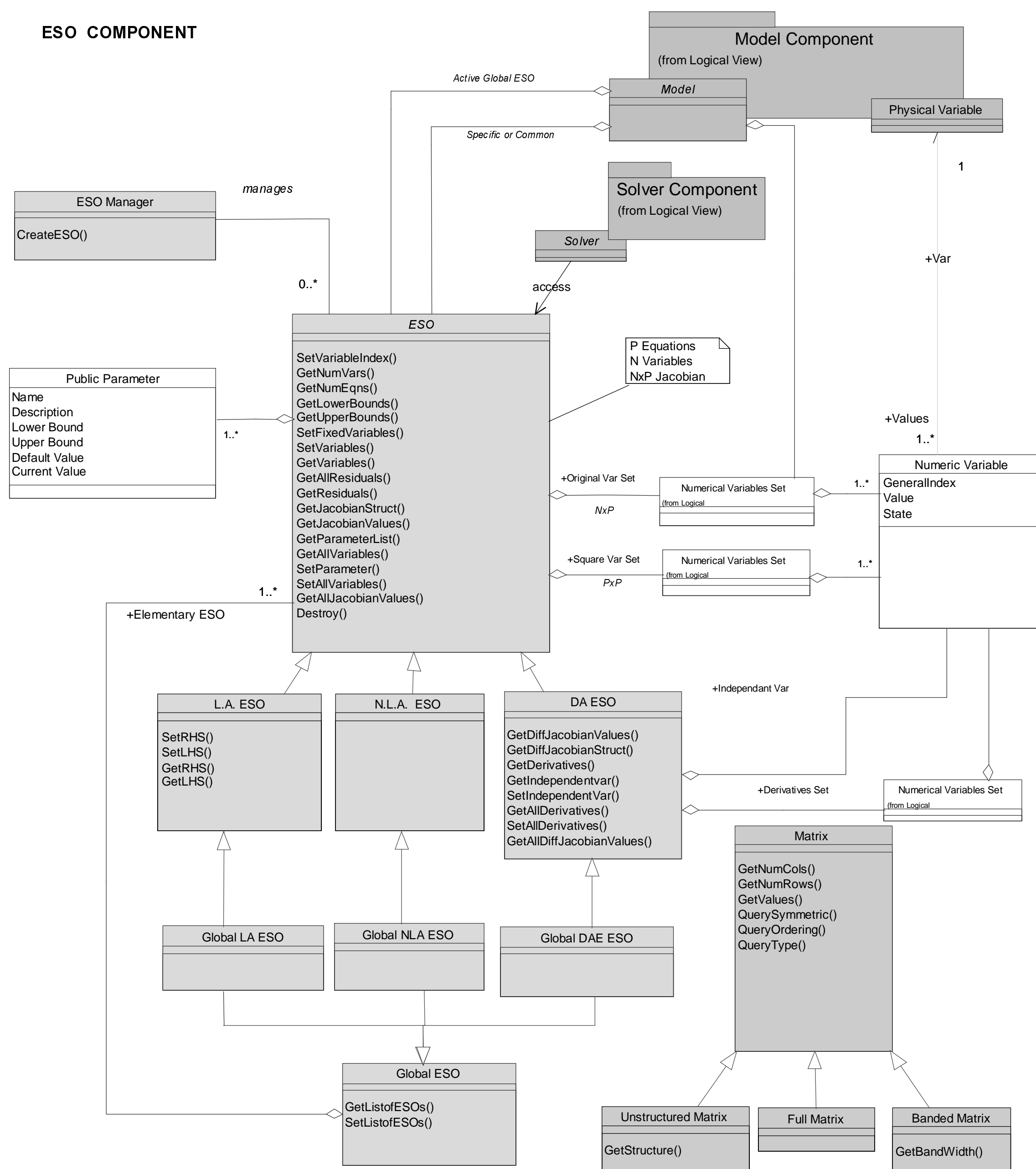



**Figure 3-1 : Class Diagram of the ESO Component**

## 3.2.4 Class Diagram of the Solver Package

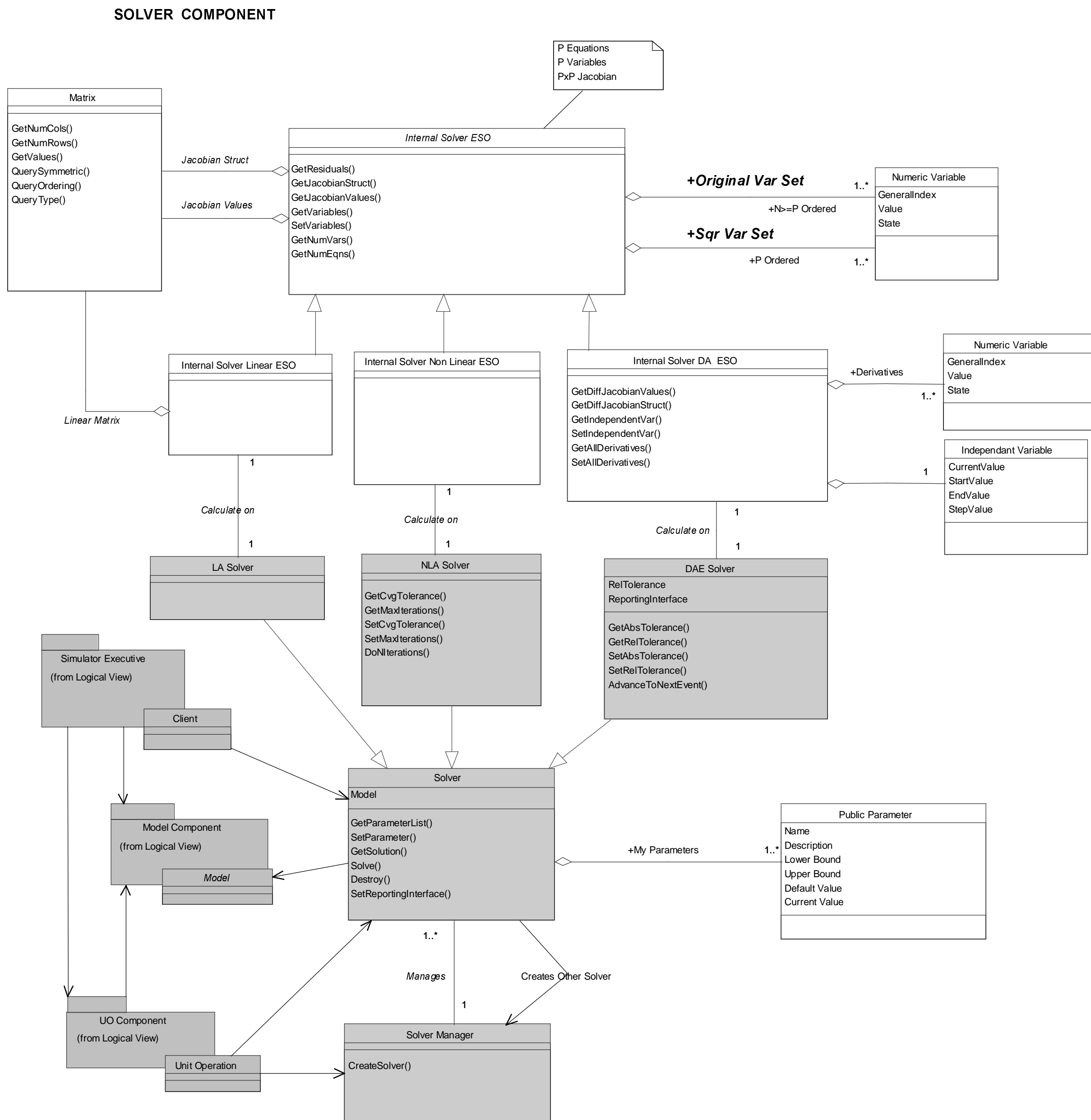



**Figure 3-1 : Class Diagram of the Solver Component**

## 3.3 Sequence Diagram

This sequence diagram describes as an example the various operations needed to configure a model using a non linear ESO and to create a solver for its resolution.

- The Client (Unit or the Simulator Executive) creates in some way, or selects a Model (top model) instance that represent the problem to be solved.

- The Client then initialises this Model. This means that the Client will have to fix some information like:

  - the current state of each STN if it is a Hierarchical Model,

  - the value of some variables (in order to get a square ESO) and,

  - the initial values of the other variables before starting the solve process.

Doing so, the Model is then able to create a global square ESO which can be used by a solver.

- The Client then creates in some way, or selects an instance of the correct type of Solver (NLASolver) needed for that problem and passes it the Model to be solved.

- As part of this creation, the NLA Solver will get all the information it needs from the Model, and his associated ESOs, such as all the initial values of the variables, number of equations, etc. As part of this creation the Solver Factory can get the list of all the parameters needed by the Solver and will set some of them to a value different from the default value.

- Then the Client will ask the Solver to solve the problem.

- In order to do that, the Solver Component (NLA Solver) will get information from the Model, and the ESO using the standard methods defined in these objects. It can also act as a client of the Solver component and create some other instances of solvers like a LASolver to solve some part of the problem.

- Then the Client will be able to get the value of the variables computed by the Solver as a solution.

**TO BE COMPLETED**

**Figure 3-1 : Sequence Diagram**

## 3.4 Collaboration diagram

This diagram is another representation of the sequence of operations showing collaboration between the different classes of object. It shows the methods that need to be defined and standardised if you want to make a component from some class or group of classes.

# TO BE COMPLETED

**Figure 3-1 : Collaboration Diagram**

## 3.5 Interface Diagrams

We have considered that the ESO is a separate component from the Unit Component, but as it acts as a server of information for the Solver Component we need to define at least its interface with it. It seems more appropriate to separate the ESO as a component from the Unit itself, since an ESO can belong to other clients as well, like the Simulator Executive or even the Solver Component if this component needs to create a specific ESO to solve DAE systems for example.

## 3.5.1 Model Interface Diagram

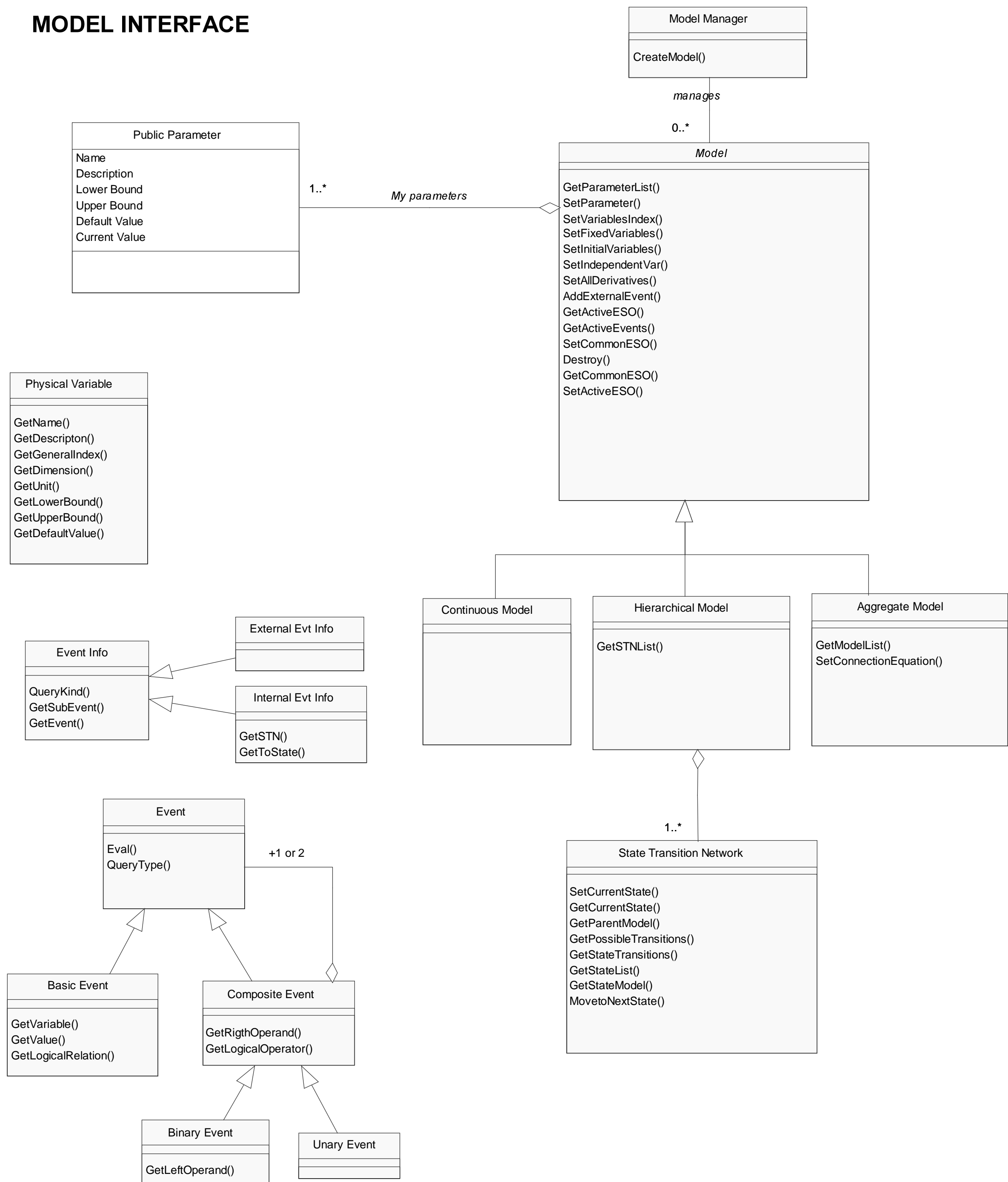



**Figure 3-1 : Model Component Interfaces**

## 3.5.2 ESO Interface Diagram

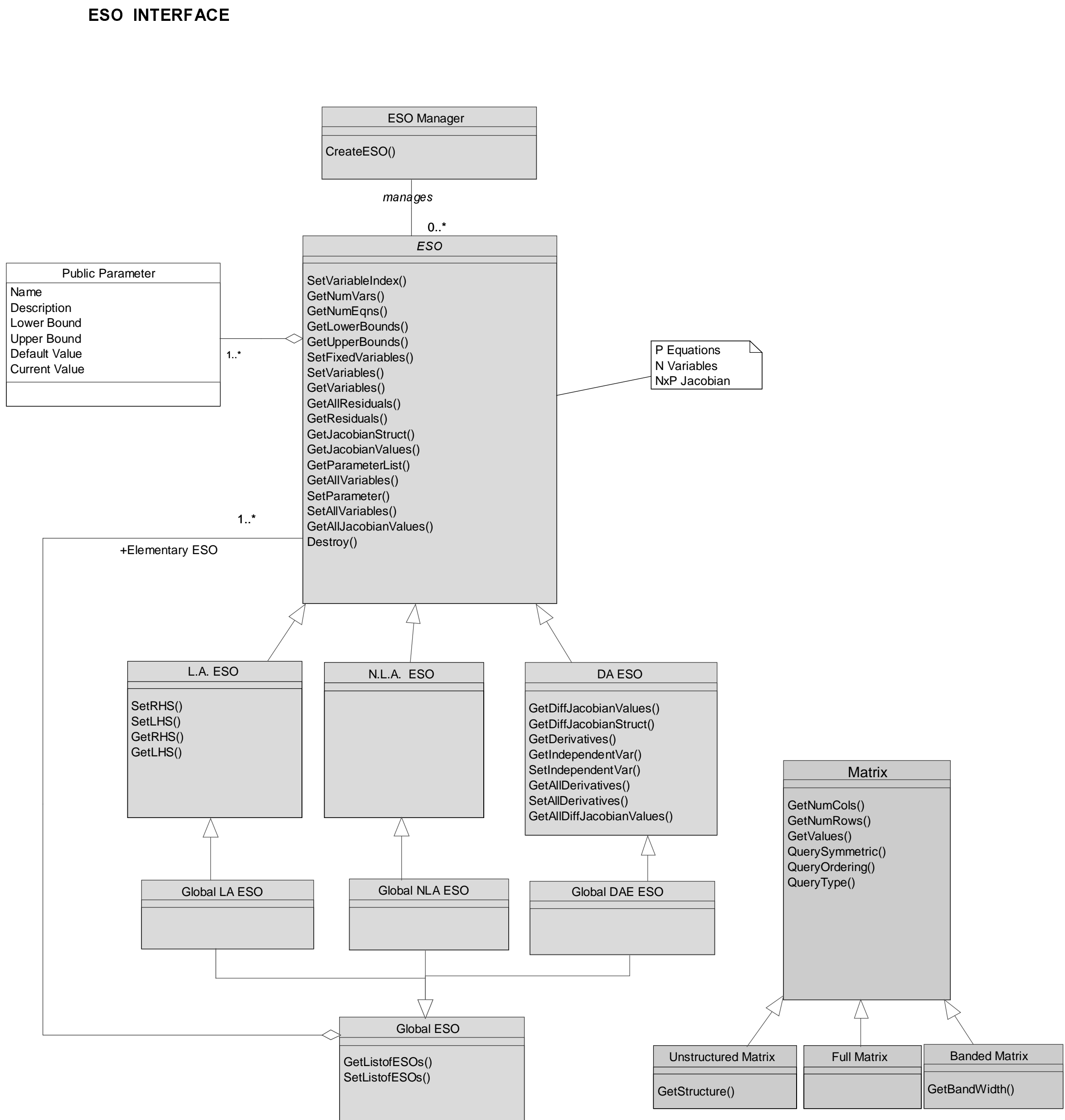



**Figure 3-1 : ESO Component Interfaces**

### 3.5.3 Solver Interface Diagram

**SOLVER INTERFACE**

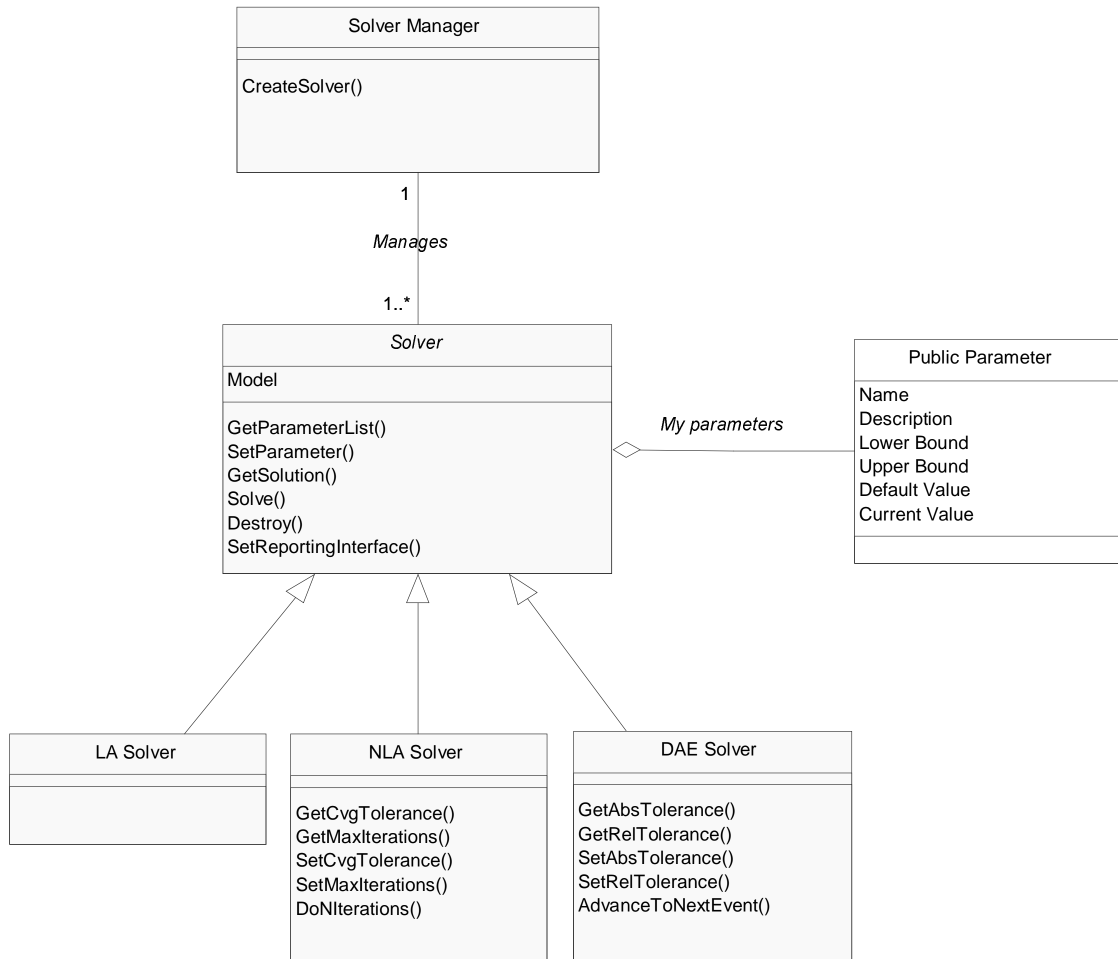



**Figure 3-1 : Solver Component Interfaces**

## 3.6 Interface Descriptions

This section details the specification of the methods appearing in the interface diagrams for the following components:

- Model Component
- ESO Component
- Solver Component

Each interface is presented together with its corresponding methods.

It should be noted that:

- Inherited methods are documented only under the parent interface which defines them.
- All methods should return a CapeError value. One role of this value is to report a successful execution: the error conditions applicable to each method will have to be defined as part of the refinement of this interface definition.
- The errors and exceptions mentioned in this specification do not pretend to be exhaustive.

Remark :

All the system errors defined as CapeError (HRESULT in DCOM or System exceptions in CORBA) are not used here. The return values described are the usual return (out retval in DCOM) from the method, and the exceptions described are only some errors that can occur during execution of the method.

We have defined a standard exception structure CapeException to handle all the possible exceptions that can be raised by each method. This structure is similar to the one defined by DCOM.

These errors can be transmitted as an argument out (in DCOM or CORBA) or as an exception (in CORBA). This is still an issue that needs to be resolved.

In defining the argument lists of the various methods, our general approach has been to use the simplest possible argument types, namely those used throughout the CAPE-OPEN project:

- CapeLong
- CapeDouble
- CapeArrayLong
- CapeArrayDouble
- CapeString
- CapeInterface

However, to provide the functionality that is necessary for our components, we have had to introduce sometimes new structures. Most of the time these structures are only applicable to one component and are part of the definition of this component. However one of these structures is common to all the components and is defined hereafter: *CapePublicParameter*, and its corresponding array type, *CapeArrayPublicParameter*.

This structure has the following members:

| | | |
|---|---|---|
| CapeString Name | : | an identifying string for this parameter |
| CapeString Description | : | a textual description of this parameter, its rôle *etc*. |
| CapeDouble LowerB (numeric parameters only) | : | the lower bound for valid values of this parameter |
| CapeDouble UpperB (numeric parameters only) | : | the upper bound for valid values of this parameter |
| CapeVariant DefaultValue | : | the default value if there is one |
| CapeVariant Value | : | the current value of this parameter |

The public parameters handled by the various interfaces presented in this document may be of any one of the types listed at the start of the section. Thus, it should be noted that the Value member of the CapePublicParameter structure presented above is a CapeVariant.

It is particularly worth noting that some algorithmic parameters are of type CapeInterface. Consider, for example, a nonlinear algebraic equation solver based on a Newton or quasi-Newton iterative scheme. An important parameter in this case would be the linear algebra solver that is used to solve the set of linear equations arising at each iteration. In our interfaces, such a parameter would be an interface to a Solver (*e.g.* ICapeNumericLASolver in the example just mentioned). Once this interface is made available, the nonlinear solver may use it to create one or more LASolvers as and when required.

### 3.6.1 Model Component

We now proceed to describe the interfaces to Models and their associated Equation Set Objects. Two other 'auxiliary' objects related to the solution of nonlinear and dynamic systems are also described, namely Event, and STN.

As we have seen, a Model may contain one or more state-transition networks (STNs). The equations in each state and the logical conditions associated with each transition in these STNs are all expressed in terms of the Model's own set of *variables,* i.e. those contained in its ESO.

Detailed information on the STNs within a Model must be obtained via a different set of methods provided by interface ICapeNumericSTN.

The Model component interfaces are:

- ICapeNumericModelManager**.** This is the interface of the Model object, which is used to represent hierarchical sets of equations, or aggregate sets of equations.
- ICapeNumericModel**.** The Model object embodies the general mathematical description of a physical system.
- ICapeNumericContinuousModel**.** This is the interface of a simple simulation model with only one ESO associated.
- ICapeNumericHierarchicalModel**.** This is the interface of a complex simulation model with a State Transition Network and multiple ESOs to pilot the simulation process.
- ICapeNumericAggregateModel**. It** allows to "concatenate" two or more previously defined models (continuous or hierarchical).
- ICapeNumericSTN**.** This is the interface which provides facilities for State Transition Networks.
- ICapeNumericEvent**.** This is the interface which provides facilities for handling Events.
- ICapeNumericBasicEvent. This is the interface which provides facilities specific to Basic Events.
- ICapeNumericCompositeEvent**.** This is the interface which provides facilities specific to Composite Events.
- ICapeNumericBinaryEvent**.** This is the interface which provides facilities specific to Binary Events.
- ICapeNumericUnaryEvent**.** This is the interface which provides facilities specific to Unary Events.
- ICapeNumericEventInfo **.** This is the interface for handling information on events.
- ICapeNumericExternalEventInfo**.** This is the interface which provides facilities specific to external events.
- ICapeNumericInternalEventInfo**.** This is the interface which provides facilities specific to internal events.

#### 3.6.1.1 Model Manager : ICapeNumericModelManager

***Inherits from:*** ICapeUtilityComponent

This is the interface of the Model object, which is used to represent hierarchical sets of equations, or aggregate sets of equations. Only one method has been defined, CreateModel.

- **CreateModel**

| **Interface Name** | **ICapeNumericModelManager** |
|---|---|
| **Method Name** | CreateModel |
| **Returns** | CapeError |

**Description**

Creates a new simulation model for a specific unit or for a complete flowsheet.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>TheTypeOfTheModel | CapeModelType | the type of the model, can be a Continuous Model, a Hierarchical Model, an Aggregate Model or one of their subtypes. |
| [out, return]<br>TheModel | ICapeNumericModel: CapeInterface | the Interface of the Model which has been created. |

**Exceptions**

To be defined (any run time error during the creation)

### 3.6.1.2 Simulation Model: ICapeNumericModel

***Inherits from:*** ICapeUtilityComponent

This interface supports the following methods:

- GetParameterList
- SetParameter
- SetVariablesIndex
- SetActiveESO
- GetActiveESO
- SetCommonESO
- GetCommonESO
- GetActiveEvents
- AddExternalEvent
- Destroy

- **GetParameterList**

| Interface Name | ICapeNumericModel |
|---|---|
| **Method Name** | GetParameterList |
| **Returns** | CapeError |

**Description**

Gets the list of all the parameters defined for this class of Model.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return] TheListOfParameters | CapeArrayNumericPublicParameter | the list of all the Public Parameters available for this class of Model. |

**Exception**

None.

- **SetParameter**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | SetParameter |
| **Returns** | CapeError |

**Description**

Sets the current value of a specific parameter to be used by the constructor of that class to create an instance of that object.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheParameterName | CapeString | the name of the parameter to be set |
| [in] TheParameterValue | CapeVariant | the value of that particular parameter |

**Exceptions**

Invalid type of the value.

Invalid parameter name.

**Remark**

This needs some experimentation, because it can be difficult for the Client to build such objects automatically and to hand them to the Solver if they are a bit complicated.

- **SetVariablesIndex**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | SetVariablesIndex |
| **Returns** | CapeError |

**Description**

Sets the general indices of the variables in this Model to establish the mapping between the list of variable in the ESO and the list of the Physical Variables. This is one way (a choice) for establishing the mapping, another way would be to reference the Physical Variables object in the Model directly.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>VarIndices | CapeArrayLong | the set of general indices for all the variables |
| [out, return]<br>TheNumberOfVars | CapeLong | the total number of variables N for this ESO |

**Exception**

Incorrect number of indices in the list (too few or too many).

- **SetActiveESO**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | SetActiveESO |
| **Returns** | CapeError |

**Description**

Sets the global ESO which is the current one depending of all the active states in the STN and all the common or specific ESOs. This needs to be done for the top level Model only.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br><br>TheActiveESO | ICapeNumericESO:CapeInterface | creates the active ESO for the top level Model. This will be some kind of Global ESO. |

**Exceptions**

To be defined.

- **GetActiveESO**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | GetActiveESO |
| **Returns** | CapeError |

**Description**

Gets the global active ESO which is the current one.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheActiveGlobalESO | ICapeNumericESO:CapeInterface | returns the ActiveGlobal ESO. |

**Exception**

No current active ESO.

- **SetCommonESO**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | SetCommonESO |
| **Returns** | CapeError |

**Description**

Assigns the common or specific ESO to this particular model.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>AnESO | ICapeNumericESO:CapeInterface | the common ESO associated with this model. |

**Exception**

None.

- **GetCommonESO**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | GetCommonESO |
| **Returns** | CapeError |

**Description**

Gets the common or specific ESO of this particular model.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br>TheCommonESO | ICapeNumericESO:ICapeInterface | the common ESO associated with this model. |

**Exception**

No common ESO associated with this model.

- **GetActiveEvents**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | GetActiveEvents |
| **Returns** | CapeError |

**Description**

Gets the list of the currently active events associated with all the current states in the model.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br><br>TheEventInfoList | CapeArrayNumericEventInfo | the list of the active events. |

**Exception**

No active state currently defined.

- **AddExternalEvent**

| **Interface Name** | **ICapeNumericModel** |
|---|---|
| **Method Name** | AddExternalEvent |
| **Returns** | CapeError |

**Description**

Adds an Event to the list of the already defined External Events for this Model.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>TheEvent | ICapeNumericEvent | the event to be added to the list. |
| [out, return]<br>AnEventInfo | ICapeNumericExternalEventInfo:CapeInterface | the External Event Info associated to that Event. |

**Exception**

None.

- **Destroy**

| Interface Name | ICapeNumericModel |
| --- | --- |
| Method Name | Destroy |
| Returns | CapeError |

**Description**

Destroys this model.

**Arguments**

None.

**Exception**

None.

- **Important Remark: Other methods**

Some other methods might be needed there that are already defined in the ESO Component like (SetFixedVars and SetAllDerivatives) just to hand values from the Client to the ESO Component.

This is needed in our architecture because we have supposed that both ESO Component and Model Component could be distributed, and only the Model knows about the ESO.

This point can be discussed later.

#### 3.6.1.3 Continuous Model: ICapeNumericContinuousModel

***Inherits from:*** ICapeNumericModel

Simple simulation model with only one ESO associated.

#### 3.6.1.4 Hierarchical Model: ICapeNumericHierarchicalModel

***Inherits from:*** ICapeNumericModel

This Model represents a complex simulation model with a State Transition Network and multiple ESOs to pilot the simulation process (mostly used for dynamic simulation with DAESO).

We have not specified here the methods that would be needed to create such a model (methods to create STN, create state, assign a model to a state, create transition, etc.), only the method needed to use such a model is defined here.

- **GetSTNList**

| **Interface Name** | **ICapeNumericHierarchicalModel** |
| --- | --- |
| **Method Name** | GetSTNList |
| **Returns** | CapeError |

**Description**

Gets the list of all the state transition networks (STNs) associated to this Hierarchical Model.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [out, return] TheSTNList | CapeArrayNumericSTN | the list of associated STNs. |

**Exception**

No STN currently associated with this Hierarchical Model.

#### 3.6.1.5 Aggregate Model: ICapeNumericAggregateModel

***Inherits from:*** ICapeNumericModel

This model allows to "concatenate" in some sense two or more previously defined models (continuous or hierarchical). This enable the creation of a complex model representing two or more units with their own variables and equations. Some equations can be added in the common ESO to represent the connections equations between these units.

- **GetModelList**

| **Interface Name** | **ICapeNumericAggregateModel** |
|---|---|
| **Method Name** | GetModelList |
| **Returns** | CapeError |

**Description**

Gets the list of all the models associated with this Aggregate Model.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br>TheListOfModels | CapeArrayNumericM odel | the list of associated models. |

**Exception**

No Model currently associated with this Aggregate Model.

- **SetConnectionEquation**

| **Interface Name** | **ICapeNumericAggregateModel** |
|---|---|
| **Method Name** | SetConnectionEquation |
| **Returns** | CapeError |

**Description**

Establish the connection between two variables that are the same in two different models of this Aggregate Model.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheInputModel | ICapeNumericModel | origin Model |
| [in] TheInputIndex | CapeLong | index of the variable |
| [in] TheOutputModel | ICapeNumericModel | destination Model |
| [in] TheOutputIndex | CapeLong | index of the variable |

**Exception**

Invalid Model or invalid index.

**Remark**

This notion of connection and its representation will probably need further investigation, there is no assumption made here on how these connection equations are represented (either by explicit equations added in the common ESO for this Aggregate Model or by identity between the two variables).

#### 3.6.1.6 State Transition Network: ICapeNumericSTN

***Inherits from:*** ICapeUtilityComponent

Eight methods are defined for this interface:

- SetCurrentState
- GetCurrentState
- GetParentModel
- GetPossibleTransitions
- GetStateTransitions
- GetStateList
- GetStateModel
- MoveToNextState

- **SetCurrentState**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | SetCurrentState |
| **Returns** | CapeError |

**Description**

Sets the value of the current state. This method can be used to set the value of the initial state or internally to switch from one state to another.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheStateName | CapeString | the name of the current state. |

**Exception**

Invalid state name.

- **GetCurrentState**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetCurrentState |
| **Returns** | CapeError |

**Description**

Gets the name of the current state for this STN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheStateName | CapeString | the name of the current state |

**Exception**

No current state defined yet.

- **GetParentModel**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetParentModel |
| **Returns** | CapeError |

**Description**

Gets the model which owns this specific STN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheParentModel | ICapeNumericModel: CapeInterface | the name of the model which owns this STN. |

**Exception**

None

- **GetPossibleTransitions**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetPossibleTransitions |
| **Returns** | CapeError |

**Description**

Gets the list of all the transitions for the current state in this STN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] TheListOfEventInfo | CapeArrayInternalEventInfo | the list of the current Internal Event Infos associated with this STN. |

**Exception**

No current state defined.

- **GetStateTransitions**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetStateTransitions |
| **Returns** | CapeError |

**Description**

Returns the names of the states which can be reached from a specified state of the network, together with the EventInfos [Events] which control each transition.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] FromState | CapeString | the name of the state. |
| [out] EventList | CapeArrayEventInfo | list of the EventInfos associated to each transition. |
| [out] StateList | CapeArrayString | list of the corresponding names of the states. |
| [out, return] NumberOfTransitions | CapeDouble | the number of transitions from that state. |

**Exceptions**

No transitions defined for this state.

Invalid state name

- **GetStateList**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetStateList |
| **Returns** | CapeError |

**Description**

Gets the list of all the states in the STN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] StateList | CapeArrayString | list of the states |

**Exceptions**

No state has been defined yet.

- **GetStateModel**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | GetStateModel |
| **Returns** | CapeError |

**Description**

Gets the model associated with a particular state.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] StateName | CapeString | the name of the state |
| [out, return] TheModel | ICapeNumericModel: CapeInterface | the Model associated with this particular state |

**Exception**

Invalid state name.

- **MoveToNextState**

| **Interface Name** | **ICapeNumericSTN** |
|---|---|
| **Method Name** | MoveToNextState |
| **Returns** | CapeError |

**Description**

Changes the current state according to the event that has fired.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] FiredEvent | ICapeNumericEventInfo:CapeInterface | the eventInfo that has triggered the change to a new state. |
| [out, return] StateName | CapeString | the name of the new current state. |

**Exception**

Invalid eventInfo passed as a parameter.

#### 3.6.1.7 Event : ICapeNumericEvent

***Inherits from:*** ICapeUtilityComponent

This object represents a condition on a variable or on a number of variables, with a boolean value. It serves two distinct roles, although the same definition is appropriate for both:

- Internal events, i.e. the transition conditions of the STNs within a Model.
- External events, i.e. those specified as stopping conditions when advancing the solution of a DAESystem (see section ICapeNumericDAESolver).

Event itself defines only a method to return its value (True or False): further information is dependent on its subtypes, and is contained in four distinct subtypes derived from it.

**CapeNumericEventType** = {BASIC, COMPOSITE, BINARY, UNARY}

The definitions make use of two other enumerated types, as follows :

**CapeLogicalRelation** = {GT, LT, GEQ, LEQ}

**CapeLogicalOperator** = {AND, OR, NOT}

- **Eval**

| **Interface Name** | **ICapeNumericEvent** |
| --- | --- |
| **Method Name** | Eval |
| **Returns** | CapeError |

**Description**

Evaluates the logical expression represented by this particular event.

**Arguments**

| Name | Type | Description |
| --- | --- | --- |
| [out, return] TheResult | CapeBoolean | True or False depending on the evaluation of the logical condition |

**Exception**

None.

- **QueryType**

| **Interface Name** | **ICapeNumericEvent** |
|---|---|
| **Method Name** | QueryType |
| **Returns** | CapeError |

**Description**

Returns the type of event involved (thus allowing the correct interface and behaviour to be determined directly rather than on a trial-and-error basis).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] AnEventType | CapeNumericEventType | returns the type of this event. |

**Exception**

None.

### 3.6.1.8 Basic Event : ICapeNumericBasicEvent

***Inherits from:*** ICapeNumericEvent

A Basic Event is a triplet of the form [variable, operator, value], like for example x5 > 1.5. We have defined the different numeric operators (>, <, >=, <=) as a CapeLogicalRelation in a typedef. Three methods are defined for this interface:

- GetVariable
- GetLogicalRelation
- GetValue

- **GetVariable**

| Interface Name | **ICapeNumericBasicEvent** |
|---|---|
| **Method Name** | GetVariable |
| **Returns** | CapeError |

**Description**

Gets the variable used in the representation of this Basic Event.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return] TheVariableIndex | CapeLong | the General index representing this variable |

**Exception**

None.

- **GetLogicalRelation**

| **Interface Name** | **ICapeNumericBasicEvent** |
|---|---|
| **Method Name** | GetLogicalRelation |
| **Returns** | CapeError |

**Description**

Gets the logical relation used in the expression of this Basic Event. This can be one of a type definition for all the supported relations (i.e. >, <, >=, <=).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheLogicalRelation | CapeLogicalRelation | the logical relation used by this basic Event. |

**Exception**

None.

- **GetValue**

| **Interface Name** | **ICapeNumericBasicEvent** |
|---|---|
| **Method Name** | GetValue |
| **Returns** | CapeError |

**Description**

Gets the value of the real constant used in the expression of that Basic Event.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br>TheValue | CapeDouble | the constant value used in the comparison. |

**Exception**

None.

### 3.6.1.9 Composite Event : ICapeNumericCompositeEvent

***Inherits from:*** ICapeNumericEvent

A Composite Event is a relation between two events links together by a logical operator (AND, OR, NOT). In the same way we have defined numeric operators, we also have defined logical operators (AND, OR, NOT) as a CapeLogicalOperator typedef.

Such a composite event can be unary or binary depending of the number of operands needed by the logical operator.

- **GetRightOperand**

| **Interface Name** | **ICapeNumericCompositeEvent** |
|---|---|
| **Method Name** | GetRightOperand |
| **Returns** | CapeError |

**Description**

Gets the right part of the Composite Event.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheRightOperand | ICapeNumericEvent:CapeInterface | the Basic or Composite Event used on the right part of the logical expression |

**Exception**

None.

- **GetLogicalOperator**

| **Interface Name** | **ICapeNumericCompositeEvent** |
|---|---|
| **Method Name** | GetLogicalOperator |
| **Returns** | CapeError |

**Description**

Gets the logical operator used in the logical expression. It must be one of the Logical Operators (AND, NOT, OR).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheLogicalOperator | CapeNumericLogicalOperator | the logical operator used in this expression. |

**Exception**

None.

### 3.6.1.10 Binary Event : ICapeNumericBinaryEvent

***Inherits from:*** ICapeNumericCompositeEvent

A Binary Event is the most common case of a Composite Event where you have a leftOperand, an Operator, and a rightOperand like in the expression A AND B.

- **GetLeftOperand**

| **Interface Name** | **ICapeNumericBinaryEvent** |
|---|---|
| **Method Name** | GetLeftOperand |
| **Returns** | CapeError |

**Description**

Gets the left logical expression in the case of binary operator.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return] TheLeftOperand | ICapeNumericEvent:CapeInterface | the left operand in the logical expression. |

**Exception**

None

### 3.6.1.11 Unary Event : ICapeNumericUnaryEvent

***Inherits from:*** ICapeNumericCompositeEvent

A Unary Event is an Event wher you do not have a leftOperand ; an example is the expression NOT A. No specific methods have been defined for this subclass.

### 3.6.1.12 Event Info : ICapeNumericEventInfo

***Inherits from:*** ICapeUtilityComponent

This object is designed as a return value from the DAESolver object, and contains information about the occurence of an Event.

The EventInfo object itself contains only a method to indicate the *kind* of event information returned (external or internal), and another to access the «sub Event» object: this is so called because even when a transition or stopping condition is specified with a composite event, the Solver is expected to return the most detailed information it can. This is likely to be a component of the original composite event.

Further detail is dependent on the kind of event information, and is contained in two distinct subtypes derived from it.

An enumerated type is needed to define the kind, as follows:

**CapeNumericEventInfoKind** = {INTERNAL, EXTERNAL}

Three methods are defined for this interface:

- QueryKind
- GetSubEvent
- GetEvent

- **QueryKind**

| **Interface Name** | **ICapeNumericEventInfo** |
|---|---|
| **Method Name** | QueryKind |
| **Returns** | CapeError |

**Description**

Returns the kind of EventInfo.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] TheKind | CapeNumericEventInfoKind | the kind of this event: internal or external. |

**Exception**

None.

- **GetSubEvent**

| **Interface Name** | **ICapeNumericEventInfo** |
| --- | --- |
| **Method Name** | GetSubEvent |
| **Returns** | CapeError |

**Description**

Provides access to the sub-event object.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [out, return]<br>AnEvent | ICapeNumericEvent:CapeInterface | the sub-event that has fired. |

**Exception**

None.

- **GetEvent**

| **Interface Name** | **ICapeNumericEventInfo** |
|---|---|
| **Method Name** | GetEvent |
| **Returns** | CapeError |

**Description**

Gets the event associated with this Event Info.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] AnEvent | ICapeNumericEvent:CapeInterface | the event associated with this EventInfo. |

**Exception**

None.

#### 3.6.1.13 External Event Info : ICapeNumericExternalEventInfo

***Inherits from:*** ICapeNumericEventInfo

When an external event occurs, we will need to know which of the stopping conditions we provided to the System has occured. This object simply adds this piece of information to that in the general EventInfo class.

#### 3.6.1.14 Internal Event Info : ICapeNumericInternalEventInfo

***Inherits from:*** ICapeNumericEventInfo

When an internal event occurs, we will generally simply have to set the state of the model as indicated, and continue the solution process. However, in order to do this or carry out some more complex action, we will require access to the STN object in which the transition «wants» to occur, as well as the target state (the current state can be obtained from the STN). This object adds these two items of information to the general EventInfo class.

- **GetSTN**

| **Interface Name** | **ICapeNumericInternalEventInfo** |
|---|---|
| **Method Name** | GetSTN |
| **Returns** | CapeError |

**Description**

Provides access to the ICapeNumericSTN object in which the state transition indicated by the EventInfo is set to occur.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheSTN | ICapeNumericSTN:CapeInterface | the STN associated with this EventInfo. |

**Exception**

None.

- **GetToState**

| **Interface Name** | **ICapeNumericInternalEventInfo** |
|---|---|
| **Method Name** | GetToState |
| **Returns** | CapeError |

**Description**

Provides the name of the state which is indicated as becoming active because of the transition condition which has became true.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] TheState | CapeString | the name of the next current state according to that Event. |

**Exception**

None.

### 3.6.2 ESO Component

This component mainly represents a rectangular system of P equations and N variables (P superior or equal to N). We have build the ESO as an independent component with all the interfaces needed between the Model Component and the Solver Component.

We have described the methods that standardise the communications with the Solver Component, and the methods that allow to map the variables used in this ESO with the variables defined by the Model.

We have defined a "Global" ESO as a subclass of the general ESO. This GlobalESO has some extra methods to set the list of the ESOs it is composed of and to manage this list.

The interfaces described are the following:

- ICapeNumericMatrix. This interface has three subtypes.
- ICapeNumericESOManager. This interface allows the creation and the management of the various ESOs.
- ICapeNumericESO. This is the interface of the Algebraic Equation Set Object.
- ICapeNumericLAESO.
- ICapeNumericNLAESO.
- ICapeNumericDAESO.
- ICapeNumericGlobalESO.
- ICapeNumericGlobalLAESO.
- ICapeNumericGlobalNLAESO.
- ICapeNumericGlobalDAESO.

### 3.6.2.1 Internal types used by this component

ICapeNumericMatrixType is an enumerated type defining the matrix types for which we have so far defined interfaces. It consists of:

(FULL, UNSTRUCTURED, BANDED)

### 3.6.2.2 Matrix interface : ICapeNumericMatrix

The **ICapeNumericMatrix** interface has the following methods:

- QueryType(): The subtype of the matrix (an ICapeNumericMatrixType value)
- GetNumCols(): number of columns in the matrix (CapeLong)
- GetNumRows(): number of rows in the matrix (CapeLong)
- GetValues(): the values of the matrix – exact semantics depend on the subtype (CapeArrayDouble)
- QuerySymmetric(): determines whether the matrix is symmetric or not (CapeBoolean)
- QueryOrdering(): determines whether values are given by row or column, for structured, unsymmetric matrices (CapeBoolean).

- **QueryType**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | QueryType |
| **Returns** | CapeError |

**Description**

This method indicates which subtype is this Matrix.

The value returned is a CapeOpen enumerated type (CapeNumericMatrixType) with three values (FULL, UNSTRUCTURED and BANDED). This allows the subtype of the matrix to be determined without trial and error.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>ASubType | CapeNumericMatrixType | returns the subtype of the matrix. |

**Exception**

None.

- **GetNumCols**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | GetNumCols |
| **Returns** | CapeError |

**Description**

Gets the number of columns in a matrix.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheNbCols | CapeLong | the number of columns. |

**Exceptions**

None.

- **GetNumRows**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | GetNumRows |
| **Returns** | CapeError |

**Description**

Gets the number of rows in a matrix.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheNbRows | CapeLong | the number of rows. |

**Exception**

None.

- **GetValues**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | GetValues |
| **Returns** | CapeError |

**Description**

This method returns an array of double values in all cases. The semantics depend on the subtype, symmetry and ordering (see QueryType and QueryOrdering).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheValues | CapeArrayLong | returns the values. |

**Exception**

None.

- **QuerySymmetric**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | QuerySymmetric |
| **Returns** | CapeError |

**Description**

Returns TRUE if this is a symmetric matrix, FALSE otherwise.

Note that NumRows must equal NumCols if this matrix is symmetric.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>ABoolean | CapeBoolean | returns TRUE or FALSE. |

**Exception**

None.

- **QueryOrdering**

| **Interface Name** | **ICapeNumericMatrix** |
|---|---|
| **Method Name** | QueryOrdering |
| **Returns** | CapeError |

**Description**

For unsymmetric matrices, some subtypes can return their values ordered either "by row" or "by column". This method indicates which type is used.

The value returned is a CapeOpen type (CapeNumericMatrixOrdering) with three values "BYROW", "BYCOL", "OTHER". ("OTHER" is for symmetric matrices where BYROW or BYCOL gives the same result, or unstructured matrices where values are returned depending on the information given by GetStructure).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>AnOrdering | CapeNumericMatrixOrdering | returns the ordering used to decode the values returned by GetValues. |

**Exception**

None.

#### 3.6.2.2.1 ICapeNumericFullMatrix

***Inherits from:*** ICapeNumericMatrix

See section 2.1.12.1 for the detailed semantics of this subtype. No further methods are defined by the interface.

#### 3.6.2.2.2 CapeNumericUnstructuredMatrix

***Inherits from:*** ICapeNumericMatrix

See section 2.1.12.2 for the detailed semantics of this subtype.

A single method is added, GetStructure(), which returns two CapeArrayLong values: RowIndices and ColumnIndices.

- **GetStructure**

| **Interface Name** | **ICapeNumericUnstructuredMatrix** |
|---|---|
| **Method Name** | GetStructure |
| **Returns** | CapeError |

**Description**

Gets the structure of the matrix (row and column indices of nonzeroes).

**Arguments**

| Name | Type | Description |
|---|---|---|
| **[out]** RowIndices | CapeArrayLong | the row list of indices. |
| **[out]** ColIndices | CapeArrayLong | the column list of indices. |

**Exception**

None.

#### 3.6.2.2.3 CapeNumericBandedMatrix

***Inherits from:*** ICapeNumericMatrix

See section 2.1.12.3 for the detailed semantics of this subtype.

A single method is added, GetBandWidth(), which returns a CapeLong value.

- **GetBandWidth**

| **Interface Name** | **ICapeNumericBandedMatrix** |
|---|---|
| **Method Name** | GetBandWidth |
| **Returns** | CapeError |

**Description**

Returns an integer N for banded matrices (no nonzero occurs more than N rows/columns from the leading diagonal).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>BandWidth | CapeLong | returns the bandwidth. |

**Exception**

None.

#### 3.6.2.3 Equation Set Object Manager interface : ICapeNumericESOManager

***Inherits from:*** ICapeUtilityComponent

This interface allows the creation and the management of the various ESOs.

There is only one instance of the ESOManager, this object knows all the ESO classes and subclasses that are available in the ESO component and will manage all the instances of ESOs that are created.

- **CreateESO**

| **Interface Name** | **ICapeNumericESOManager** |
|---|---|
| **Method Name** | CreateESO |
| **Returns** | CapeError |

**Description**

Creates a new ESO. This can be done also by instantiating some known subclasses of ESO.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>TheTypeOfESO | CapeESOType | the type of ESO to be created may be LA, NLA, DAE or Global, or some other subclasses of these classes. |
| [out, return]<br>AnESO | ICapeNumericESO:CapeInterface | the Interface of the ESO which has been created. |

**Exception**

To be defined (any run time error during the creation)

### 3.6.2.4 Equation Set Object (ESO) interface : ICapeNumericESO

***Inherits from:*** ICapeUtilityComponent

This is the interface of the Equation Set Object which in the most general case represents a set of equations of the form :

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

In general, a set described by an ESO can be rectangular, *i.e.* the number of variables does not have to be the same as the number of equations[9].

The variables in an ESO are characterised by their current values (that can be changed via the provided interface), and also lower and upper bounds. Usually, these bounds relate to the domain of definition of the equations[10] and/or physical reality[11]. For this reason, any attempt to set one or more variables to values outside these bounds is considered to be illegal and will, therefore, be rejected.

The equations in an ESO are assumed to be *sparse*, *i.e.* any given equation will involve only a subset of the variables in the ESO. Consequently, only a (usually small) subset of the partial derivatives $\partial f / \partial x$ are going to be nonzero for *any* set of values of the variables *x*. The *sparsity pattern* of the ESO refers to the number of such nonzero elements, and the row *i* (*i.e.* equation $f_i$) and column *j* (*i.e.* variable $x_j$) to which each such nonzero corresponds. The way in which information on this structure is defined is entirely analogous to that for linear systems.

The interface defined in this section provided mechanisms for obtaining information on the current values and bounds of the variables $x$, as well as the sparsity pattern of the ESO. It also allows the modification of the variable values, and the computation of the values («residuals») of the equations $f(x)$ for the current values of $x$ and of the nonzero elements of the matrix $\partial f / \partial x$ (the so-called «Jacobian» matrix).

Finally, we note that CAPE-OPEN does *not* define any standard mechanisms or interfaces for the *construction* of ESOs. These are left at the discretion of implementers.

---

[9] Of course, any ESOs that is to be solved using the nonlinear algebraic solver interfaces described in section Numerical Solver Component must be *square*, *i.e.* it must have the same number of equations and variables.

[10] For instance, an equation involving a term $\sqrt{1-x}$ is undefined for any value of variable $x$ exceeding unity; thus, $x$ is subject to an upper bound of 1.0.

[11] For instance, variables representing molar fractions must stay between a lower bound of 0 and an upper bound of 1.

The methods of the ICapeNumericESO interface are:

- GetParameterList
- SetParameter
- GetNumVars
- GetNumEqns
- SetFixedVars
- SetAllVariables
- SetVariables
- GetAllVariables
- GetVariables
- GetAllResiduals
- GetResiduals
- GetJacobianStruct
- GetAllJacobianValues
- GetJacobianValues
- Destroy

- **GetParameterList**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetParameterList |
| **Returns** | CapeError |

**Description**

Gets the list of all the parameters defined for this ESO class.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheListOfParameters | CapeArrayNumericPublicParameter | the list of all the Public Parameter available for this class of ESO. |

**Exception**

None.

- **SetParameter**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | SetParameter |
| **Returns** | CapeError |

**Description**

Sets the current value of a specific parameter to be used by the constructor of that class to create an instance of that object.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheParameterName | CapeString | the name of the parameter to be set. |
| [in] TheParameterValue | CapeVariant | the value of that particular parameter. |

**Exceptions**

Invalid type of the value.

Invalid parameter name.

- **GetNumVars**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetNumVars |
| **Returns** | CapeError |

**Description**

Gets the number of variables of this ESO. In the case of a "Global" ESO (built by a complex model), it will return the total number of variables in the Global ESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br><br>TheNumberOfVars | CapeLong | the total number of variables N for this ESO |

**Exception**

No set of variables associated yet with this ESO.

- **GetNumEqns**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetNumEqns |
| **Returns** | CapeError |

**Description**

Gets the number of equations in this ESO. In the case of a Global ESO, it will return the total number of equations for this Global ESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br><br>TheNumberOfEqns | CapeLong | the total number of equations P for this ESO. |

**Exception**

None.

- **SetFixedVars**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | SetFixedVars |
| **Returns** | CapeError |

**Description**

Sets the value of some variables and marks these variables as fixed.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] VarIndices | CapeArrayLong | the indices of the variables we wish to set. |
| [in] VarValues | CapeArrayDouble | the values of the variables we wish to set. |

**Exceptions**

No set of variables associated yet with this ESO.

Index out of range for some variables.

Not the same length in the two arrays.

- **SetAllVariables**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | SetAllVariables |
| **Returns** | CapeError |

**Description**

Sets the value of all variables of this ESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] VarValues | CapeArrayDouble | the values of all the variables we wish to set. |

**Exceptions**

No set of variables associated yet with this ESO.

Too many values in the array (extra values can be ignored).

Not enough values in the array (values missing can be set to 0).

- **SetVariables**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | SetVariables |
| **Returns** | CapeError |

**Description**

Sets the value of some variables.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] VarIndices | CapeArrayLong | the indices of the variables we wish to set. |
| [in] VarValues | CapeArrayDouble | the values of all the variables we wish to set. |

**Exceptions**

No set of variables associated yet with this ESO.

Index out of range for some variables.

Not the same length in the two arrays.

- **GetAllVariables**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetAllVariables |
| **Returns** | CapeError |

**Description**

Gets the value of all variables.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return] VarValues | CapeArrayDouble | the values of all the variables. |

**Exception**

No set of variables associated yet with this ESO.

- **GetVariables**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetVariables |
| **Returns** | CapeError |

**Description**

Gets the value of a subset of the variables.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] VarIndices | CapeArrayLong | the indices of the variables we wish to get. |
| [out, return] VarValues | CapeArrayDouble | the values of the subset of variables. |

**Exceptions**

No set of variables associated yet with this ESO.

Index out of range for some variables.

- **GetAllResiduals**

| **Interface Name** | **ICapeNumericESO** |
| --- | --- |
| **Method Name** | GetAllResiduals |
| **Returns** | CapeError |

**Description**

Gets the value of all the residuals.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [out, return]<br>AllResiduals | CapeArrayDouble | the values of all the residuals for all the equations. |

**Exception**

Variables not initialised.

- **GetResiduals**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetResiduals |
| **Returns** | CapeError |

**Description**

Gets the value of a subset of the residuals.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>Indices | CapeArrayLong | the indices of the equations we wish to get the residuals from. |
| [out, return]<br>TheResiduals | CapeArrayDouble | the values of the residuals for the requested equations. |

**Exception**

Invalid indices for the equations.

**Remark**

All the residuals are evaluated at the current variables values.

- **GetJacobianStruct**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetJacobianStruct |
| **Returns** | CapeError |

**Description**

Returns a matrix object which contains information on the structure of the Jacobian matrix. The GetValues method of this object will provide values encoded as follows :

- -1.0 indicates an entry which cannot be computed by the ESO.
- 0.0 indicates an entry which will **always** be zero.

Any other value indicates a nonzero, computable entry.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheMatrix | ICapeNumericMatrix: CapeInterface | a matrix or any matrix subclass with the Jacobian structure. |

**Exception**

None.

- **GetAllJacobianValues**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetAllJacobianValues |
| **Returns** | CapeError |

**Description**

Returns a matrix object whose GetValues method will provide the Jacobian values at the ESO's current variable values each time it is called (the values of entries indicated as uncomputable in the matrix returned by GetJacobianStruct will be meaningless, but the call to GetValues will not cause an error simply because such entries exist).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br><br>TheMatrix | ICapeNumericMatrix:<br>CapeInterface | a Matrix object yielding the Jacobian values. |

**Exception**

None.

- **GetJacobianValues**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | GetJacobianValues |
| **Returns** | CapeError |

**Description**

Gets the values of selected entries of the Jacobian, at the current variable values of the ESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheElementIndices | CapeArrayLong | the indices of selected elements. The semantics are those of the matrix's GetValues method. |
| [out, return] TheValues | CapeArrayDouble | the values of the requested elements. |

**Exceptions**

Some of the requested Jacobian values cannot be computed.

Indices out of range.

- **Destroy**

| **Interface Name** | **ICapeNumericESO** |
|---|---|
| **Method Name** | Destroy |
| **Returns** | CapeError |

**Description**

Deletes the ESO Component and all the objects associated with this particular ESO Component.

**Arguments**

None.

**Exception**

None.

### 3.6.2.5 Linear Analysis ESO interface : ICapeNumericLAESO

***Inherits from:*** ICapeNumericESO

Only a few methods have been defined for this interface, more could be added. Four methods have been defined so far:

- SetRHS
- SetLHS
- GetRHS
- GetLHS

- **SetRHS**

| **Interface Name** | **ICapeNumericLAESO** |
|---|---|
| **Method Name** | SetRHS |
| **Returns** | CapeError |

**Description**

Sets the values of the right hand side vector.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>Values | CapeArrayDouble | the values of the RHS to be set. |

**Exceptions**

None.

- **SetLHS**

| Interface Name | ICapeNumericLAESO |
|---|---|
| Method Name | SetLHS |
| Returns | CapeError |

**Description**

Sets the left hand side values of the linear system, i.e. the matrix values. The input argument is interpreted with the same semantics as the array returned when the GetValues method is called.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>Matrix | ICapeNumericMatrix | the values of the LHS to be set. |

**Exception**

None.

- **GetRHS**

| **Interface Name** | **ICapeNumericLAESO** |
|---|---|
| **Method Name** | GetRHS |
| **Returns** | CapeError |

**Description**

Gets the values most recently set for the right hand side vector of this linear system.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>Values | CapeArrayDouble | the values of the RHS. |

**Exceptions**

No values have been set.

- **GetLHS**

| **Interface Name** | **ICapeNumericLAESO** |
|---|---|
| **Method Name** | GetLHS |
| **Returns** | CapeError |

**Description**

Gets the left hand side values of this linear system.

**Note**: the result should be the same as what is returned by calling the GetValues method of the Matrix object returned by GetAllJacobianValues.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>Matrix | ICapeNumericMatrix | the current values of the LHS. |

**Exceptions**

No values have been set..

#### 3.6.2.6 Non Linear Analysis ESO interface : ICapeNumericNLAESO

***Inherits from:*** ICapeNumericESO

There is no special method right now for this interface.

### 3.6.2.7 Differential Analysis ESO interface : ICapeNumericDAESO

***Inherits from:*** ICapeNumericESO

This is the interface of the Differential-Algebraic Equation Set Object which represents a (generally rectangular) set of differential-algebraic equations of the form:

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

where $t$ is the independent variable and $x(t)$ is a vector of dependent variables. Also $\dot{x}$ denotes the derivatives $dx/dt$. We note that, in general, the quantities $\dot{x}$ will appear in the system for only a subset of the dependent variables $x$. This subset of $x$ are often referred to as the «*differential variables*» while the rest are the «*algebraic variables*». Of course, *all* these variables are functions of the independent variable $t$.

It is worth clarifying the semantic interpretation of the methods that are inherited by this interface from ICapeNumericESO:

- GetNumVars must return the length of the vector x
- SetVariables and GetVariables relate only to the vector $x$.
- All the methods associated with the Jacobian (GetJacobianStruct and GetJacobianValues) relate to $\partial f / \partial x$.
- The equation residuals and Jacobian are evaluated at the current values of $x, \dot{x}$ and $t$.

The methods defined in this section introduce equivalent functionality for accessing and altering information pertaining to $f(x, \dot{x}) = 0$. They also provide mechanisms for accessing and altering the value of the independent variable $t$. The defined methods are:

- SetAllDerivatives
- GetAllDerivatives
- GetDerivatives
- GetDiffJacobianStruct
- GetAllDiffJacobianValues
- GetDiffJacobianValues
- SetIndependentVar
- GetIndependentVar

- **SetAllDerivatives**

| **Interface Name** | **ICapeNumericDAESO** |
|---|---|
| **Method Name** | SetAllDerivatives |
| **Returns** | CapeError |

**Description**

Sets the numerical value of all the derivatives. The length of the array supplied must equal the number of variables: values for derivatives which do not appear in any of the equations can be ignored.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] VarValues | CapeArrayDouble | the values of the derivatives. |

**Exceptions**

No set of variables associated yet with this ESO.

Number of values provided not equal to the number of variables.

Not the same length in the two arrays.

- **GetAllDerivatives**

| **Interface Name** | **ICapeNumericDAESO** |
|---|---|
| **Method Name** | GetAllDerivatives |
| **Returns** | CapeError |

**Description**

Gets the values of the derivatives for all the variables. The length of the array returned will be equal to the number of variables, but the values of derivatives which do not appear in the equation system may be meaningless.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br><br>TheValues | CapeArrayDouble | the values of all the derivatives. |

**Exception**

No set of variables associated yet with this ESO.

- **GetDerivatives**

| **Interface Name** | **ICapeNumericDAESO** |
| --- | --- |
| **Method Name** | GetDerivatives |
| **Returns** | CapeError |

**Description**

Gets the value of a subset of the derivatives.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [in]<br>TheIndices | CapeArrayLong | the indices of the variables whose derivatives we wish to get. |
| [out, return]<br>TheValues | CapeArrayDouble | the values of the subset of derivatives. |

**Exceptions**

No set of variables associated yet with this ESO.

Index out of range for some variables.

- **GetDiffJacobianStruct**

| **Interface Name** | **ICapeNumericDAESO** |
|---|---|
| **Method Name** | GetDiffJacobianStruct |
| **Returns** | CapeError |

**Description**

Returns a matrix object which contains information on the structure of the differential Jacobian matrix. The GetValues method of this object will provide values encoded as follows:

- -1.0 indicates an entry which cannot be computed by the ESO.
- 0.0 indicates an entry which will **always** be zero.
- Any other value indicates a nonzero, computable entry.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>Matrix | ICapeNumericMatrix | full or sparse matrix |

**Exception**

None.

- **GetAllDiffJacobianValues**

| **Interface Name** | **ICapeNumericDAESO** |
|---|---|
| **Method Name** | GetAllDiffJacobianValues |
| **Returns** | CapeError |

**Description**

Returns a matrix object whose GetValues method will provide the differential Jacobian values at the ESO's current variable values each time it is called (the values of entries indicated as uncomputable in the matrix returned by GetDiffJacobianStruct will be meaningless, but the call to GetValues will not cause an error simply because such entries exist).

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheMatrix | ICapeNumericMatrix | a Matrix object yielding the differential Jacobian values. |

**Exception**

None.

- **GetDiffJacobianValues**

| **Interface Name** | **ICapeNumericDAESO** |
| --- | --- |
| **Method Name** | GetDiffJacobianValues |
| **Returns** | CapeError |

**Description**

Gets the values of selected entries of the differential Jacobian, at the current variable values of the ESO.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [in] TheElementIndices | CapeArrayLong | the indices of selected elements. The semantics are those of the matrix's GetValues method. |
| [out, return] TheValues | CapeArrayDouble | the values of the requested elements. |

**Exception**

None.

- **SetIndependentVar**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | SetIndependentVar |
| **Returns** | CapeError |

**Description**

Sets the value of the independent variable in the DAESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] IndVarValue | CapeDouble | the value of the independent variable |

**Exceptions**

None.

- **GetIndependentVar**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | GetIndependentVar |
| **Returns** | CapeError |

**Description**

Gets the current value of the independent variable in the DAESO.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>IndVarValue | CapeDouble | the value of the independent variable |

**Exceptions**

None.

#### 3.6.2.8 Global ESO interface : ICapeNumericGlobalESO

***Inherits from:*** ICapeUtilityComponent

This interface is there to allow concatenation of multiples ESOs in the case we want to aggregate multiple ESOs (in the case of a flowsheet) in a single ESO.

It has specific methods for that.

- **SetListOfESOs**

| **Interface Name** | **ICapeNumericGlobalESO** |
|---|---|
| **Method Name** | SetListOfESOs |
| **Returns** | CapeError |

**Description**

Sets the list of all the ESOs included in this Global ESO.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>ListOfESOs | CapeArrayNumericESO | the list of the ESOs the Global ESO is composed of. |

**Exception**

Incompatible ESO types.

- **GetListOfESOs**

| **Interface Name** | **ICapeNumericGlobalESO** |
|---|---|
| **Method Name** | GetListOfESOs |
| **Returns** | CapeError |

**Description**

Gets the list of all the ESOs included in this Global ESO.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br>ListOfESOs | CapeArrayNumericESO | the list of the ESOs the Global ESO is composed of. |

**Exception**

None.

#### 3.6.2.9 Global LAESO interface : ICapeNumericGlobalLAESO

***Inherits from:*** ICapeNumericGlobalESO and ICapeNumericLAESO

This GlobalESO is a list of LAESOs and inherits from LAESO and GlobalESO.

#### 3.6.2.10 Global NLAESO interface : ICapeNumericGlobalNLAESO

***Inherits from:*** ICapeNumericGlobalESO and ICapeNumericNLAESO

This GlobalESO is a list of NLAESOs and inherits from NLAESO and GlobalESO.

#### 3.6.2.11 Global DAESO interface : ICapeNumericGlobalDAESO

***Inherits from:*** ICapeNumericGlobalESO and ICapeNumericDAESO

This GlobalESO is a list of DAESOs and inherits from DAESO and GlobalESO.

### 3.6.3 Solver Component

This part describes all the interfaces and their associated methods for the Solver Component itself. The different interfaces are:

- ICapeNumericSolverManager. This interface creates an instance of the Solver Component.
- ICapeNumericSolver. This interface provides facilities that are common to the different kinds of solvers.
- ICapeNumericLASolver. This interface provides facilities which are specific to Solvers of Linear Algebraic equation systems.
- ICapeNumericNLASolver. This interface provides facilities which are specific to Solvers of Non Linear Algebraic equation systems.
- ICapeNumericDAESolver. This interface provides facilities which are specific to Solvers of Differential Algebraic equation systems.

#### 3.6.3.1 Solver Manager interface : ICapeNumericSolverManager

***Inherits from:*** ICapeUtilityComponent

We first need to have a factory to create an instance of the Solver Component for a specific ESO from a specific type, either linear, non linear or differential.

- **CreateSolver**

| **Interface Name** | **ICapeNumericSolverManager** |
|---|---|
| **Method Name** | CreateSolver |
| **Returns** | CapeError |

**Description**

Creates a specific Solver Component of the type appropriate to the input argument.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>TheModel | ICapeNumericModel : CapeInterface | the reference of the Model to be solved. |
| [in]<br>theSolverType | CapeSolverType | the type of the Solver to be created. |
| [out, return]<br>theSolver | ICapeNumericSolver:CapeInterface | the reference of the Solver Component created. |

**Exceptions**

The total number of equations in the Model differs from the number of variables.

Other possible errors.

### 3.6.3.2 Numeric Solver interface : ICapeNumericSolver

***Inherits from:*** ICapeUtilityComponent

This interface exists to provide facilities for identifying the various algorithmic parameters (*e.g.* convergence accuracy, integration error tolerances *etc.*) that are recognised by a numerical solver, and for altering their values if necessary.

We have also grouped here all the methods that are common to the different kinds of solvers. Six methods have been defined:

- 

GetParameterList

- SetParameter
- Solve
- GetSolution
- Destroy
- SetReportingInterface

- **GetParameterList**

| **Interface Name** | **ICapeNumericSolver** |
|---|---|
| **Method Name** | GetParameterList |
| **Returns** | CapeError |

**Description**

Gets the list of all the parameters defined for this Solver class.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheListOfParameters | CapeArrayNumericPublicParameter | the list of all the Public Parameter available for this class of Solver. |

**Exception**

None.

- **SetParameter**

| Interface Name | ICapeNumericSolver |
|---|---|
| Method Name | SetParameter |
| Returns | CapeError |

**Description**

Sets the current value of a specific parameter to be used by the constructor of that class to create an instance of that object.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in] TheParameterName | CapeString | the name of the parameter to be set. |
| [in] TheParameterValue | CapeVariant | the value of that particular parameter. |

**Exceptions**

Invalid type of the value.

Invalid parameter name.

.

- **Solve**

| Interface Name | ICapeNumericSolver |
|---|---|
| Method Name | Solve |
| Returns | CapeError |

**Description**

Attempts to solve the system of equations defined within the Model associated with this solver instance. Return will occur in different circumstances:

- The attempt succeeds.
- The solver gives up.
- An unexpected problem («exception») arises.

If the underlying Model contains one or more STNs, the Solver may consider a switch to a different set of active states necessary to finding a solution. In this case, it may use the MoveToNextState method of one or more of the Model's STNs during the computation. The numerical values held in the model on return should be consistent with the set of states which are active on return.

Distinction between the first three types of return will be made by examining the return argument (one component of this will be a general OK/Error flag, where Error indicates an exception, while another will be a Success/Failure flag).

**Notes** : The initial guesses for this solution will be the current variable values of the Model.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return] TheStatus | CapeLong | a code indicating if a solution has been found and under which conditions (Ex : MaxIterations reached or some more complex conditions especially in the case of a DAE Solver). This will need further investigations. |

**Exceptions**

Various exceptions could appear in this method, like LA Solver not solvable, or initials values not set, etc.

- **GetSolution**

| **Interface Name** | **ICapeNumericSolver** |
|---|---|
| **Method Name** | GetSolution |
| **Returns** | CapeError |

**Description**

Gets all the values of the variables that solve this System.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheValues | CapeArrayDouble | the array of the P free variables in the system. |

**Exception**

No solution is available: either the Solve method has not been called, or it has ended with an error.

- **Destroy**

| **Interface Name** | **ICapeNumericSolver** |
| --- | --- |
| **Method Name** | Destroy |
| **Returns** | CapeError |

**Description**

Deletes the Solver Component and all the objects associated to this particular Solver Component.

**Arguments**

None.

**Exception**

None.

- **SetReportingInterface**

| **Interface Name** | **ICapeNumericSolver** |
|---|---|
| **Method Name** | SetReportingInterface |
| **Returns** | CapeError |

**Description**

Sets the reference to an object in charge of managing some reporting at each step of the process.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>ReportingInterface | CapeInterface | the object reference of the reporting object. |

**Exception**

None.

**Remark**

The reporting interface will be called by the Solver:

- Immediately on entry to AdvanceToNextEvent.
- When the independent variable reaches a value specified by it (see interface).
- Before and after all discontinuities handled internally be the Solver.
- Immediately before return from AdvanceToNextEvent.

It is its own responsibility to decide what to do and which information needs to be displayed.

### 3.6.3.3 Numeric LA Solver interface : ICapeNumericLASolver

***Inherits from:*** ICapeNumericSolver

No specific methods have been defined for this kind of solver and no assumption has been made either for the representation of the vector and matrix of the system. It is left open to the implementation.

We have assumed that the Solve method get the [A] matrix and the [B] vector of the [A][X]=[B] system using the already defined methods.

The [A] matrix is given by the GetJacobianValues method of the ESO and the [B] vector is equal to minus the GetResiduals method with all the variables set to zero. The [X] vector result is given by the GetSolution method.

#### 3.6.3.4 Numeric NLA Solver interface : ICapeNumericNLASolver

***Inherits from:*** ICapeNumericSolver

In this section we define the interfaces related to the solution of sets of nonlinear algebraic equations.

This interface defines methods for the identification and setting of parameters that will occur in *all* CAPE-OPEN compliant nonlinear algebra components. A small number of such generic parameters have been identified; separate methods are defined for obtaining information on, and changing the value of each such parameter. Five methods have been defined:

- SetCvgTolerance
- GetCvgTolerance
- SetMaxIterations
- GetMaxIterations
- DoNIterations

- **SetCvgTolerance**

| **Interface Name** | **ICapeNumericNLASolver** |
|---|---|
| **Method Name** | SetCvgTolerance |
| **Returns** | CapeError |

**Description**

Sets the convergence tolerance to be used in solving a nonlinear system. The precise interpretation of this parameter will depend on individual implementations; the nature of the convergence criterion used by nonlinear solvers is not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>CTolValue | CapeDouble | the convergence tolerance value. |

**Exception**

None.

- **GetCvgTolerance**

| **Interface Name** | **ICapeNumericNLASolver** |
| --- | --- |
| **Method Name** | GetCvgTolerance |
| **Returns** | CapeError |

**Description**

Gets information on the convergence tolerance to be used in solving a nonlinear system, as well as its current value. The precise interpretation of this parameter will depend on individual implementations; the nature of the convergence criterion used by nonlinear solvers is not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [out, return]<br>CTolValue | CapeDouble | the convergence tolerance value. |

**Exception**

No value set for the convergence tolerance.

- **SetMaxIterations**

| **Interface Name** | **ICapeNumericNLASolver** |
| --- | --- |
| **Method Name** | SetMaxIterations |
| **Returns** | CapeError |

**Description**

Sets the maximum number of iterations to be used in solving a nonlinear system. The precise interpretation of this parameter will depend on individual implementations; the nature of what constitutes an «iteration» used by nonlinear solvers is not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [in]<br>MaxItsValue | CapeLong | the maximum number of iterations. |

**Exception**

None.

.

- **GetMaxIterations**

| **Interface Name** | **ICapeNumericNLASolver** |
|---|---|
| **Method Name** | GetMaxIterations |
| **Returns** | CapeError |

**Description**

Gets information on the maximum number of iterations to be used in solving a nonlinear system, as well as its current value. The precise interpretation of this parameter will depend on individual implementations; the nature of what constitutes an «iteration» used by nonlinear solvers is not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>MaxItsValue | CapeLong | the maximum number of iterations. |

**Exception**

No value set for the maximum number of iterations.

- **DoNIterations**

| **Interface Name** | **ICapeNumericNLASolver** |
|---|---|
| **Method Name** | DoNIterations |
| **Returns** | CapeError |

**Description**

Perform N iterations on the nonlinear algebra problem. The possible returns are the same as for Solve, except that in this case «OK+Failure» may merely indicate that more iterations are needed.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in]<br>NbIteration | CapeLong | the number of iterations to be performed. |
| [out, return]<br>ReturnCode | CapeLong | depends of the conditions reached (0=normal return, or 1=convergence tolerance, 2=max iterations reached before N iterations). |

**Exceptions**

Same exceptions as in the Solve method. Various exceptions could appear in this method, like LA Solver not solvable, or initials values not set, etc.

### 3.6.3.5 Numeric DAE Solver interface : ICapeNumericDAESolver

***Inherits from:*** ICapeNumericSolver

In this section, we describe the interfaces related to the solution of differential-algebraic equation systems.

This interface defines methods for the identification and setting of parameters that will occur in *all* CAPE-OPEN compliant differential-algebraic components. A small number of such generic parameters have been identified; separate methods are defined for obtaining information on, and changing the value of each such parameter. The defined methods are:

- SetRelTolerance
- GetRelTolerance
- SetAbsTolerance
- GetAbsTolerance
- AdvanceToNextEvent

- **SetRelTolerance**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | SetRelTolerance |
| **Returns** | CapeError |

**Description**

Sets the relative tolerance values to be used in performing local error tests while solving a DAE system. The precise interpretation of this parameter will depend on individual implementations; the exact nature of the error measure used (*e.g.* local truncation error, local error *etc.*) and the way in which this is estimated are not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] RelTolValues | CapeArrayDouble | the relative tolerance values. |

**Exception**

Incorrect size of the array.

- **GetRelTolerance**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | GetRelTolerance |
| **Returns** | CapeError |

**Description**

Gets information on the relative tolerance to be used in performing local error tests while solving a DAE system, as well as its current value. The precise interpretation of this parameter will depend on individual implementations; the exact nature of the error measure used (*e.g.* local truncation error, local error *etc.*), and the way in which this is estimated, are not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [out, return]<br>TheRelTolValues | CapeArrayDouble | the relative tolerance values. |

**Exception**

Tolerance not set.

- **SetAbsTolerance**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | SetAbsTolerance |
| **Returns** | CapeError |

**Description**

Sets the absolute tolerance to be used in performing local error tests while solving the DAE system. The precise interpretation of this parameter will depend on individual implementations; the exact nature of the error measure used (*e.g.* local truncation error, local error *etc.*), and the way in which this is estimated, are not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] AbsTolValues | CapeArrayDouble | the absolute tolerance values. |

**Exception**

Incorrect size of the array.

- **GetAbsTolerance**

| **Interface Name** | **ICapeNumericDAESolver** |
| --- | --- |
| **Method Name** | GetAbsTolerance |
| **Returns** | CapeError |

**Description**

Gets information on the absolute tolerance to be used in performing local error tests while solving a DAE system, as well as the current value of this parameter. The precise interpretation of this parameter will depend on individual implementations; the exact nature of the error measure used (*e.g.* local truncation error, local error *etc.*) and the way in which this is estimated are not defined by CAPE-OPEN.

**Arguments**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| [out, return]<br>TheAbsTolValues | CapeArrayDouble | the absolute tolerance values. |

**Exception**

Tolerance not set.

- **AdvanceToNextEvent**

| **Interface Name** | **ICapeNumericDAESolver** |
|---|---|
| **Method Name** | AdvanceToNextEvent |
| **Returns** | CapeError |

**Description**

Advances the solution of the DAESO with respect to its independent variable until some Event(s) occurs, or an error occurs in the solution process.

**Note :** For dynamic problems, we cannot be certain that the Solver will be able to identify precisely the single Event which causes termination. Thus we allow it to provide a list of Event objects, together with independent variable values which «bracket» the termination point.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] EndConditions | CapeArrayNumericEventInfo | the list of stopping conditions for this call. |
| [out] TimeBefore | CapeDouble | the independent variable value at the beginning of the internal step. |
| [out] TimeAfter | CapeDouble | the independent variable value at the end of the internal step. |
| [out, return] ListOfEvents | CapeArrayNumericEventInfo | a list indicating the cause(s) of termination. |

**Exceptions**

To be defined later.

# 4 Interface specifications

This section contains the CORBA IDL instructions. They are compilable files that you can directly use for producing CAPE-OPEN compliant components for Solvers. No COM IDL was developed so far.

## 4.1 CORBA IDL

### 4.1.1 Common definitions

#### 4.1.1.1 Utility Definitions

```
//
// CORBA CapeUtilityDefinitions.idl
//

#ifndef CapeUtilityDefinitions_idl
#define CapeUtilityDefinitions_idl


//#include corba.h //only for C++

// Primitive common data types

typedef boolean CapeBoolean;
typedef sequence<CapeBoolean> CapeArrayBoolean;

typedef char CapeChar;
typedef sequence<CapeChar> CapeArrayChar;

typedef short CapeShort;
typedef sequence<CapeShort> CapeArrayShort;

typedef long CapeLong;
typedef sequence<CapeLong> CapeArrayLong;

typedef float CapeFloat;
typedef sequence<CapeFloat> CapeArrayFloat;

typedef double CapeDouble;
typedef sequence<CapeDouble> CapeArrayDouble;

typedef string CapeString;
typedef sequence<CapeString> CapeArrayString;

typedef any CapeVariant;
typedef sequence<CapeVariant> CapeArrayVariant;

typedef Object CapeInterface;
typedef sequence<CapeInterface> CapeArrayInterface;

// Exceptions

exception CapeException {
      CapeLong type;
      CapeLong minus;
```

```
      CapeLong completed;
      CapeString idlmethod;
      CapeString explanation;
};

#endif
```

#### 4.1.1.2 Utility Component

```
//
// CORBA CapeUtilityComponent.idl
//

#include "CapeUtilityDefinitions.idl"


#ifndef CapeUtilityComponent_idl
#define CapeUtilityComponent_idl

module CapeUtilityComponent {

// CORBA Utility Component interface : ICapeUtilityComponent

interface ICapeUtilityComponent {

      CapeString GetVersionNumber() raises(CapeException);

      CapeString GetComponentName() raises(CapeException);

      CapeString GetComponentDescription() raises(CapeException);

};
typedef sequence<ICapeUtilityComponent> CapeArrayUtilityComponent;


// Definition of Numeric structures that should be in Common

struct CapePublicParameter {
      CapeString  name;
      CapeString  description;
      CapeLong    lowerBound;
      CapeLong    upperBound;
      CapeVariant defaultValue;
      CapeVariant currentValue;
};
typedef sequence<CapePublicParameter> CapeArrayPublicParameter;


}; // end of CapeUtilityComponent module

#endif
```

### 4.1.2 Model Component

```
//
// CORBA CapeNumericModelComponent.idl
//
```

```
#include "CapeUtilityComponent.idl"
#include "CapeNumericESOComponent.idl"

#ifndef CapeNumericModelComponent_idl
#define CapeNumericModelComponent_idl


module CapeNumericModelComponent {

// Some enumerated types

typedef enum ModelTypes {
      CONTINUOUS,
      HIERARCHICAL,
      AGGREGATE
} CapeModelType;

typedef enum EventTypes {
      BASIC,
      COMPOSITE,
      BINARY,
      UNARY
} CapeEventType;

typedef enum EventInfoKinds {
      EXTERNAL,
      INTERNAL
} CapeEventInfoKind;

typedef enum LogicalRelations {
      GEQ,
      LEQ,
      GT,
      LT
} CapeLogicalRelation;

typedef enum LogicalOperators {
      AND,
      OR,
      NOT
} CapeLogicalOperator;


interface ICapeNumericModelManager;
typedef sequence<ICapeNumericModelManager> CapeArrayNumericModelManager;
interface ICapeNumericModel;
typedef sequence<ICapeNumericModel> CapeArrayNumericModel;
interface ICapeNumericContinuousModel;
typedef sequence<ICapeNumericContinuousModel>
CapeArrayNumericContinuousModel;
interface ICapeNumericHierarchicalModel;
typedef sequence<ICapeNumericHierarchicalModel>
CapeArrayNumericHierarchicalModel;
interface ICapeNumericAggregateModel;
typedef sequence<ICapeNumericAggregateModel>
CapeArrayNumericAggregateModel;
interface ICapeNumericSTN;
typedef sequence<ICapeNumericSTN> CapeArrayNumericSTN;
interface ICapeNumericEvent;
typedef sequence<ICapeNumericEvent> CapeArrayNumericEvent;
```

```
interface ICapeNumericBasicEvent;
typedef sequence<ICapeNumericBasicEvent> CapeArrayNumericBasicEvent;
interface ICapeNumericCompositeEvent;
typedef sequence<ICapeNumericCompositeEvent>
CapeArrayNumericCompositeEvent;
interface ICapeNumericBinaryEvent;
typedef sequence<ICapeNumericBinaryEvent> CapeArrayNumericBinaryEvent;
interface ICapeNumericUnaryEvent;
typedef sequence<ICapeNumericUnaryEvent> CapeArrayNumericUnaryEvent;
interface ICapeNumericEventInfo;
typedef sequence<ICapeNumericEventInfo> CapeArrayNumericEventInfo;
interface ICapeNumericExternalEventInfo;
typedef sequence<ICapeNumericExternalEventInfo>
CapeArrayNumericExternalEventInfo;
interface ICapeNumericInternalEventInfo;
typedef sequence<ICapeNumericInternalEventInfo>
CapeArrayNumericInternalEventInfo;


// ***** CORBA Model Manager interface : ICapeNumericModelManager

interface ICapeNumericModelManager :
CapeUtilityComponent::ICapeUtilityComponent {

      ICapeNumericModel CreateModel(in CapeModelType typeOfTheModel)
                              raises (CapeException);

};


// ***** CORBA Simulation Model interface : ICapeNumericModel

interface ICapeNumericModel :
CapeUtilityComponent::ICapeUtilityComponent {

      CapeUtilityComponent::CapeArrayPublicParameter GetParameterList()
                              raises (CapeException);
      void SetParameter(in CapeString parameterName, in CapeVariant
parameterValue)
                              raises (CapeException);
      CapeLong SetVariableIndex(in CapeArrayLong varIndices)
                              raises (CapeException);
      CapeNumericESOComponent::ICapeNumericESO SetActiveESO()
                              raises (CapeException);
      CapeNumericESOComponent::ICapeNumericESO GetActiveESO()
                              raises (CapeException);
      void SetCommonESO(in CapeNumericESOComponent::ICapeNumericESO
anESO)
                              raises (CapeException);
      CapeNumericESOComponent::ICapeNumericESO GetCommonESO()
                              raises (CapeException);
      CapeArrayNumericEventInfo GetActiveEvents()
                              raises (CapeException);
      ICapeNumericExternalEventInfo AddExternalEvent(in
ICapeNumericEvent anEvent)
                              raises (CapeException);
      void Destroy();
};
```

```
// ***** CORBA Continuous Simulation Model interface :
ICapeNumericContinuousModel

interface ICapeNumericContinuousModel : ICapeNumericModel {

};


// ***** CORBA Hierarchical Simulation Model :
ICapeNumericHierarchicalModel

interface ICapeNumericHierarchicalModel : ICapeNumericModel {

      CapeArrayNumericSTN GetSTNList() raises (CapeException);
};


// ***** CORBA Aggregate Simulation Model : ICapeNumericAggregateModel

interface ICapeNumericAggregateModel : ICapeNumericModel {

      CapeArrayNumericModel GetModelList() raises (CapeException);
      void SetConnectionEquation(in ICapeNumericModel inputModel,
                                       in CapeLong inputIndex,
                                       in ICapeNumericModel
outputModel,
                                       in CapeLong outputIndex)
                              raises (CapeException);
};


// ***** CORBA State Transition Network : ICapeNumericSTN

interface ICapeNumericSTN : CapeUtilityComponent::ICapeUtilityComponent
{

      void SetCurrentState(in CapeString theStateName) raises
(CapeException);
      CapeString GetCurrentState()  raises (CapeException);
      ICapeNumericModel GetParentModel() raises (CapeException);
      CapeArrayNumericInternalEventInfo GetPossiblesTransitions()
                               raises (CapeException);
      CapeDouble GetStateTransitions(in CapeString fromState,
                                             out
CapeArrayNumericEventInfo eventList,
                                             out CapeArrayString
stateList)
                              raises (CapeException);
      CapeArrayString GetStateList() raises (CapeException);
      ICapeNumericModel GetStateModel(in CapeString stateName)
                              raises (CapeException);
      CapeString MoveToNextState(in ICapeNumericEventInfo firedEvent)
                              raises (CapeException);
};


// ***** CORBA Event : ICapeNumericEvent

interface ICapeNumericEvent :
CapeUtilityComponent::ICapeUtilityComponent {
```

```
      CapeBoolean eval() raises (CapeException);
      CapeEventType QueryType() raises (CapeException);
};


// ***** CORBA Basic Event : ICapeNumericBasicEvent

interface ICapeNumericBasicEvent : ICapeNumericEvent {

      CapeLOng GetVariable() raises (CapeException);
      CapeLogicalRelation GetLogicalRelation() raises (CapeException);
      CapeDouble GetValue() raises (CapeException);
};


// ***** CORBA Composite Event : ICapeNumericCompositeEvent

interface ICapeNumericCompositeEvent :ICapeNumericEvent {

      ICapeNumericEvent GetRightOperand() raises (CapeException);
      CapeLogicalOperator GetLogicalOperator()
                  raises (CapeException);
};


// CORBA Binary Event : ICapeNumericBinaryEvent

interface ICapeNumericBinaryEvent : ICapeNumericCompositeEvent {

      ICapeNumericEvent GetLeftOperand() raises (CapeException);
};


// ***** CORBA Unary Event : ICapeNumericUnaryEvent

interface ICapeNumericUnaryEvent : ICapeNumericCompositeEvent {

};


// ***** CORBA Event Info : ICapeNumericEventInfo

interface ICapeNumericEventInfo :
CapeUtilityComponent::ICapeUtilityComponent {

      CapeEventInfoKind QueryKind() raises (CapeException);
      ICapeNumericEvent GetSubEvent() raises (CapeException);
      ICapeNumericEvent GetEvent() raises (CapeException);
};


// ***** CORBA External Event Info : ICapeNumericEventInfo

interface ICapeNumericExternalEventInfo : ICapeNumericEventInfo {

};


// ***** CORBA Internal Event Info : ICapeNumericEventInfo

interface ICapeNumericInternalEventInfo : ICapeNumericEventInfo {
```

```
      ICapeNumericSTN GetSTN() raises (CapeException);
      CapeString GetToState()  raises (CapeException);
};

#endif

}; // end CapeNumericModelComponent module
```

### 4.1.3 ESO Component

```
//
// CORBA CapeNumericESOComponent.idl
//

#include "CapeUtilityComponent.idl"

#ifndef CapeNumericESOComponent_idl
#define CapeNumericESOComponent_idl

module CapeNumericESOComponent {

typedef enum ESOtypes {
      LA,
      NLA,
      DAE,
      GLOBAL
} CapeESOType;


typedef enum MatrixTypes {
      FULL,
      UNSTRUCTURED,
      BANDED
} CapeMatrixType;

typedef enum QueryOrderings {
      ROW,
      COLUMN,
      OTHER
} CapeMatrixOrdering;


interface ICapeNumericMatrix;
typedef sequence<ICapeNumericMatrix> CapeArrayNumericMatrix;
interface ICapeNumericFullMatrix;
typedef sequence<ICapeNumericFullMatrix> CapeArrayNumericFullMatrix;
interface ICapeNumericUnstructuredMatrix;
typedef sequence<ICapeNumericUnstructuredMatrix>
CapeArrayNumericUnstructuredMatrix;
interface ICapeNumericBandedMatrix;
typedef sequence<ICapeNumericBandedMatrix> CapeArrayNumericBandedMatrix;
interface ICapeNumericESOManager;
typedef sequence<ICapeNumericESOManager> CapeArrayNumericESOManager;
interface ICapeNumericESO;
typedef sequence<ICapeNumericESO> CapeArrayNumericESO;
interface ICapeNumericLAESO;
typedef sequence<ICapeNumericLAESO> CapeArrayNumericLAESO;
interface ICapeNumericNLAESO;
typedef sequence<ICapeNumericNLAESO> CapeArrayNumericNLAESO;
interface ICapeNumericDAESO;
typedef sequence<ICapeNumericDAESO> CapeArrayNumericDAESO;
interface ICapeNumericGlobalESO;
typedef sequence<ICapeNumericGlobalESO> CapeArrayNumericGlobalESO;
interface ICapeNumericGlobalLAESO;
typedef sequence<ICapeNumericGlobalLAESO> CapeArrayNumericGlobalLAESO;
interface ICapeNumericGlobalNLAESO;
typedef sequence<ICapeNumericGlobalNLAESO> CapeArrayNumericGlobalNLAESO;
interface ICapeNumericGlobalDAESO;
```

```
typedef sequence<ICapeNumericGlobalDAESO> CapeArrayNumericGlobalDAESO;


// ***** CORBA Matrix interface : ICapeNumericMatrix

interface ICapeNumericMatrix  :
CapeUtilityComponent::ICapeUtilityComponent {

      CapeLong          GetNumRows() raises(CapeException);
      CapeLong          GetNumCols() raises(CapeException);
      CapeBoolean       QuerySymmetric() raises(CapeException);
      CapeMatrixOrdering      QueryOrdering() raises(CapeException);
      CapeMatrixType    QueryType() raises(CapeException);
      CapeArrayDouble GetValues() raises(CapeException);

};


// ***** CORBA FullMatrix interface : ICapeNumericFullMatrix

interface ICapeNumericFullMatrix : ICapeNumericMatrix {

};


// ***** CORBA Unstructured Matrix interface :
ICapeNumericUnstructuredMatrix

interface ICapeNumericUnstructuredMatrix : ICapeNumericMatrix {

      void GetStructure(out CapeArrayDouble rowIndices,
                                      out CapeArrayDouble colIndices)
                  raises(CapeException);

};


// ***** CORBA Banded Matrix interface : ICapeNumericBandedMatrix

interface ICapeNumericBandedMatrix : ICapeNumericMatrix {

      CapeLong GetBandWidth() raises(CapeException);

};


// ***** CORBA  ESO Manager interface : ICapeNumericESOManager

interface ICapeNumericESOManager :
CapeUtilityComponent::ICapeUtilityComponent {

      ICapeNumericESO CreateESO(in CapeESOType typeOfESO)
                              raises (CapeException);

};


// ***** CORBA Equation Set Object (ESO) interface : ICapeNumericESO

interface ICapeNumericESO :  CapeUtilityComponent::ICapeUtilityComponent
{
```

```
	CapeUtilityComponent::CapeArrayPublicParameter GetParameterList()
				raises (CapeException);
	void SetParameter(in CapeString parameterName, in CapeVariant
parameterValue)
				raises (CapeException);
	CapeLong GetNumVars() raises(CapeException);
	CapeLong GetNumEqns() raises(CapeException);
	void SetFixedVariables(in CapeArrayLong varIndices,
				in CapeArrayDouble varValues)
				raises(CapeException);
	void SetAllVariables(in CapeArrayDouble varValues)
				raises(CapeException);
	void SetVariables(in CapeArrayLong varIndices,
				in CapeArrayDouble varValues)
				raises(CapeException);
	CapeArrayDouble GetAllVariables() raises(CapeException);
	CapeArrayDouble GetVariables(in CapeArrayLong varIndices)
				raises(CapeException);
	CapeArrayDouble GetAllResiduals() raises(CapeException);
	CapeArrayDouble GetResiduals(in CapeArrayLong eqnIndices)
				raises(CapeException);
	ICapeNumericMatrix GetJacobianStruct() raises(CapeException);
	ICapeNumericMatrix GetAllJacobianValues()
				raises(CapeException);
	CapeArrayDouble GetJacobianValues(in CapeArrayLong indices)
				raises(CapeException);
	void SetVariablesIndex(in CapeArrayLong varIndices);
	CapeArrayDouble GetLowerBounds() raises(CapeException);
	CapeArrayDouble GetUpperBounds() raises(CapeException);
	void Destroy();

};


// ***** CORBA  Linear Algebraic ESO (LA ESO) interface :
ICapeNumericESO

interface ICapeNumericLAESO : ICapeNumericESO {

	void  SetLHS(in ICapeNumericMatrix values) raises (CapeException);
	void  SetRHS(in CapeArrayDouble values) raises (CapeException);
	ICapeNumericMatrix	GetLHS() raises (CapeException);
	CapeArrayDouble   GetRHS() raises (CapeException);

};


// ***** CORBA  Non Linear Algebraic ESO (NLA ESO) interface :
ICapeNumericESO

interface ICapeNumericNLAESO : ICapeNumericESO {

};


// ***** CORBA  Differential Algebraic ESO (DAESO) interface :
ICapeNumericDAESO

interface ICapeNumericDAESO : ICapeNumericESO {
```

```
    void SetAllDerivatives(in CapeArrayDouble varValues)
                raises(CapeException);
    CapeArrayDouble GetAllDerivatives() raises(CapeException);
    CapeArrayDouble GetDerivatives(in CapeArrayLong varIndices)
                            raises(CapeException);
    ICapeNumericMatrix GetDiffJacobianStruct() raises(CapeException);
    ICapeNumericMatrix GetAllDiffJacobianValues()
                              raises(CapeException);
    CapeArrayDouble GetDiffJacobianValues(in CapeArrayLong indices)
                              raises(CapeException);
    void SetIndependentVar(in CapeDouble indVar)
raises(CapeException);
    CapeDouble GetIndependentVar() raises(CapeException);

};


// ***** CORBA Global ESO (GlobalESO) interface : ICapeNumericGlobalESO

interface ICapeNumericGlobalESO : ICapeNumericESO {

    void SetListOfESOs(in CapeArrayNumericESO listOfESOs);
    CapeArrayNumericESO GetListOfESOs();

};


// ***** CORBA Global LAESO (GlobalLAESO) interface :
ICapeNumericGlobalLAESO

interface ICapeNumericGlobalLAESO : ICapeNumericLAESO,
ICapeNumericGlobalESO {

};


// ***** CORBA Global NLAESO (GlobalNLAESO) interface :
ICapeNumericGlobalNLAESO

interface ICapeNumericGlobalNLAESO : ICapeNumericNLAESO,
ICapeNumericGlobalESO {

};


// ***** CORBA Global DAESO (GlobalDAESO) interface :
ICapeNumericGlobalDAESO

interface ICapeNumericGlobalDAESO : ICapeNumericDAESO,
ICapeNumericGlobalESO {

};


#endif

}; // end CapeNumericESOComponent module
```

### 4.1.4 Solver Component

```
//
// CORBA CapeNumericSolverComponent.idl
//

#include "CapeNumericModelComponent.idl"

#ifndef ICapeNumericSolverComponent_idl
#define ICapeNumericSolverComponent_idl

module CapeNumericSolverComponent {


typedef enum Solvertypes {
      LA,
      NLA,
      DAE
} CapeSolverType;

interface ICapeNumericSolverManager;
typedef sequence<ICapeNumericSolverManager>
CapeArrayNumericSolverManager;
interface ICapeNumericSolver;
typedef sequence<ICapeNumericSolver> CapeArrayNumericSolverComponent;
interface ICapeNumericLASolver;
typedef sequence<ICapeNumericLASolver> CapeArrayNumericLASolver;
interface ICapeNumericNLASolver;
typedef sequence<ICapeNumericNLASolver> CapeArrayNumericNLASolver;
interface ICapeNumericDAESolver;
typedef sequence<ICapeNumericDAESolver> CapeArrayNumericDAESolver;


// ***** CORBA Solver  interface : ICapeNumericSolverManager

interface ICapeNumericSolverManager :
CapeUtilityComponent::ICapeUtilityComponent {

      ICapeNumericSolver CreateSolver(
                                    in CapeSolverType type,
                                    in
CapeNumericModelComponent::ICapeNumericModel theModel)
                  raises (CapeException);

};


//***** CORBA Solver interface : ICapeNumericSolver

interface ICapeNumericSolver :
CapeUtilityComponent::ICapeUtilityComponent {

      CapeUtilityComponent::CapeArrayPublicParameter GetParameterList()
                              raises (CapeException);
      void SetParameter(in CapeString parameterName, in CapeVariant
parameterValue)
                              raises (CapeException);
      CapeLong Solve() raises (CapeException);
      CapeArrayDouble GetSolution() raises (CapeException);
      CapeInterface SetReportingInterface() raises (CapeException);
      void Destroy();
```

```
};

// ***** CORBA LA Solver interface : ICapeNumericLASolver

interface ICapeNumericLASolver : ICapeNumericSolver  {

};

// ***** CORBA NLASolver interface : ICapeNumericNLASolver

interface ICapeNumericNLASolver : ICapeNumericSolver  {

      void SetCvgTolerance(in CapeDouble cvgValue)
                  raises (CapeException);
      CapeDouble GetCvgTolerance() raises (CapeException);
      void SetMaxIterations(in CapeLong maxIteration)
                  raises (CapeException);
      CapeLong GetMaxIterations() raises (CapeException);
      CapeLong DoNIteration(in CapeLong nbIterations)
                        raises (CapeException);
};

// ***** CORBA DAESolver interface : ICapeNumericSolverDAESolver

interface ICapeNumericDAESolver : ICapeNumericSolver  {

      void SetRelTolerance(in CapeArrayDouble relTolValue)
                        raises (CapeException);
      CapeArrayDouble GetRelTolerance()
                        raises (CapeException);
      void SetAbsTolerance(in CapeArrayDouble absTolValues)
                         raises (CapeException);
      CapeArrayDouble GetAbsTolerance()
                        raises (CapeException);
      CapeNumericModelComponent::CapeArrayNumericEventInfo
AdvanceToNextEvent(
                  in
CapeNumericModelComponent::CapeArrayNumericEventInfo endConditions,
                  out CapeDouble timeBefore,
                  out CapeDouble timeAfter)
             raises (CapeException);

};

#endif

}; // End of CapeNumericSolverSolver
```

## 4.2 COM IDL

This section needs to be completed.

### 4.2.1 Common definitions

### 4.2.2 Model Component

### 4.2.3 ESO Component

### 4.2.4 Solver Component

# 5 Notes on analysis and interface specifications

We tried to summarise in this part of the document some of the remaining issues and problems that need further investigations and discussions.

## 5.1 Differences between CORBA and COM

As already mentioned before in this document there is some important differences between COM and CORBA. One of them is the way exceptions are handled in CORBA and in COM.

This could lead to different interfaces.

## 5.2 Public Parameter

In order to allow some customisation of each component when an instance of this component is created we have defined a Public Parameter structure which can be used either by a Solver, a Model or an ESO. Each parameter value can be any object instance.

This could be difficult to implement, and will need some communication with the user (through a user interface or some language specification) to get the correct values for each of these parameters.

- **GetParameterList**

| Interface Name | |
|---|---|
| **Method Name** | GetParameterList |
| **Returns** | CapeError |

**Description**

Gets the list of all the public parameters defined for a given class (ESO, Model, Solver, etc.). This is a Class method that allows to customise the objects that will be created by the constructor of that Class.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [out, return]<br><br>TheListOfParameters | CapeArrayNumericPublicParameter | the list of all the Public Parameters available for this class. |

**Exceptions**

None.

- **SetParameter**

| **Interface Name** | |
|---|---|
| **Method Name** | SetParameter |
| **Returns** | CapeError |

**Description**

Sets the current value of a parameter in the parameter list.

**Arguments**

| **Name** | **Type** | **Description** |
|---|---|---|
| [in] TheParameterName | CapeString | the name of the parameter to be set. |
| [in] TheParameterValue | CapeVariant | the value of that particular parameter. |

**Exceptions**

Invalid type of the value.

Invalid parameter name.

## 5.3 Reporting

We have supposed that there is somewhere an object capable of displaying the information we want to display during the process of solving the DAE System. It is the responsibility of the solver component to inform this object each time new pieces of information have been calculated, but no communication scheme has been defined yet. Here is a proposal for an IcapeNumericReport Interface. It has not been integrated in the main body of the interface definitions since it remains a general issue within the CAPE-OPEN project.

This interface requires the following enumerated type :

> CapeNumericReportReason = (INITIAL, INDVARREACHED, BEFOREDISC, AFTERDISC, FINAL)

- **ReportModelValues**

| **Interface Name** | **ICapeNumericReport** |
|---|---|
| **Method Name** | ReportModelValues |
| **Returns** | CapeError |

**Description**

Gives a reporting routine an opportunity to extract and display values from the Model, and specify the next call.

**Arguments**

| Name | Type | Description |
|---|---|---|
| [in]<br>TheModel | CapeInterface | handle to the model which the Solver is solving |
| [in]<br>ReasonForCall | CapeNumericReportReason | why the reporting routine has been called (see list under the SetReportingInterface method). |
| [out]<br>NextTime | CapeDouble | the value of the independent variable at which the Solver should next be called (unless an Event occurs) |

**Exceptions**

None.

## 5.4 Thermo and Physical Property Packages

In order to do some calculation, the ESO package may need to access the thermodynamic package. This has not been modelled yet. It means that the ESO should know in some way how to access to the THRM package.

The problem is the same as for the reporting.