CAPE-OPEN

Delivering the power of component software
and open standard interfaces
in Computer-Aided Process Engineering

Methods & Tools Integrated Guidelines

www.colan.org

ARCHIVAL INFORMATION

Filename	Methods & Tools Integrated Guidelines.doc
Authors	CO-LaN consortium
Status	Internal Draft
Date	August 2003
Version	version 1
Number of pages	84
Versioning	version 1, Jean-Pierre Belaud, Bertrand Braunschweig, August 2003
Additional material	
Web location	www.colan.org
Implementation specifications version	CAPE-OPENv1-0-0.idl (CORBA) CAPE-OPENv1-0-0.zip and CAPE-OPENv1-0-0.tlb (COM)
Comments	This document gathers, updates and completes a set of former documents coming from the former Methods and Tools group. So additionally to the above authors, many other peoples from diverse companies have contributed to this document.

IMPORTANT NOTICES

Disclaimer of Warranty

CO-LaN documents and publications include software in the form of *sample code.* Any such software described or provided by CO-LaN --- in whatever form --- is provided "as-is" without warranty of any kind. CO-LaN 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 Laboratories Network --- remains with you.

CO-LaN is a non for profit organization established under French law of 1901.

Trademark Usage

Many of the designations used by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in CO-LaN 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.

SUMMARY

This document gathers the set of Methods and Tools (M&T) recommendations and guidelines for the CAPE-OPEN (CO) standard.

It presents the conclusions of the Methods and Tools Group, in charge of making the choices on methods and software tools for the CAPE-OPEN initiative. The main recommendations are to use the UML notation and to implementation the open interfaces both for OLE/COM and CORBA.

It provides guidelines for developing CAPE-OPEN interface specifications. It addresses important software analysis and procedural issues and makes specific recommendations in order that CO interface developers can create uniform release design and documents.

It gives architectural and technical information for developing CO compliant components.

ACKNOWLEDGEMENTS

CONTENTS

LIST OF FIGURES

1. Introduction

This document integrates a number of Methods and Tools (M&T) guidelines developed during the CAPE-OPEN (CO) and Global CAPE-OPEN (GCO) projects. Its aim is to offer to the CAPE-OPEN interfaces and components developers all the generic information needed. It is built an evolving document, as M&T participants will continue to offer new material through CO-LaN organisation.

The document is not a summary of CAPE-OPEN interfaces. All vertical CAPE-OPEN interface specifications (e.g. for physical properties calculations, unit operations, solvers etc.) are defined in other documents. Instead, the M&T material provided here is seen as reference material for further developing and implementing those domain-specific interfaces.

The subjects considered here are:

- ***Concepts and Definitions*** that defines the different types of interfaces and organises all CO materials in a documentation set.
- ***Selected Methods and Tools*** that provides general decisions on methods and tools and on the work process for developing CO interfaces.
- ***Software engineering elements*** that gives elementary recommendations for designing uniform CO interfaces.
- ***General architectural and technical issues*** that discusses on the general architecture of CO standard.
- ***COM-specific architectural and technical issues*** that concerns the Microsoft COM platform.
- ***CORBA-specific architectural and technical issues*** that is related to OMG CORBA platform.
- ***Common interfaces*** that explains the objective of common interfaces and introduces the current interfaces.

1.1 Intended Audience

- ***developers of CAPE-OPEN compliant software components*** will find technical information about the generic elements of the CAPE-OPEN standard
- ***developers of CAPE-OPEN standard interface specifications*** will find generic information needed to produce additional specifications
- ***software and CAPE specialists*** interested in knowing more about the CAPE-OPEN standard will find a summary of the generic information that supports the CAPE-OPEN standard.

1.2 What you should read

If you are a developers of CAPE-OPEN compliant software components, you should read sections 2, 3, 6, 7 or 8 and 9

If you are a developer of CAPE-OPEN standard interface specifications, you should read all the sections;

If you are just curious about the Common Services and Generic Guidelines for CAPE-OPEN, you should read sections 2, 3 and 4.

1.3 CAPE-OPEN history and the first recommendations documents

As a reminder, the next figure shows the different projects that led to the CAPE-OPEN standard.

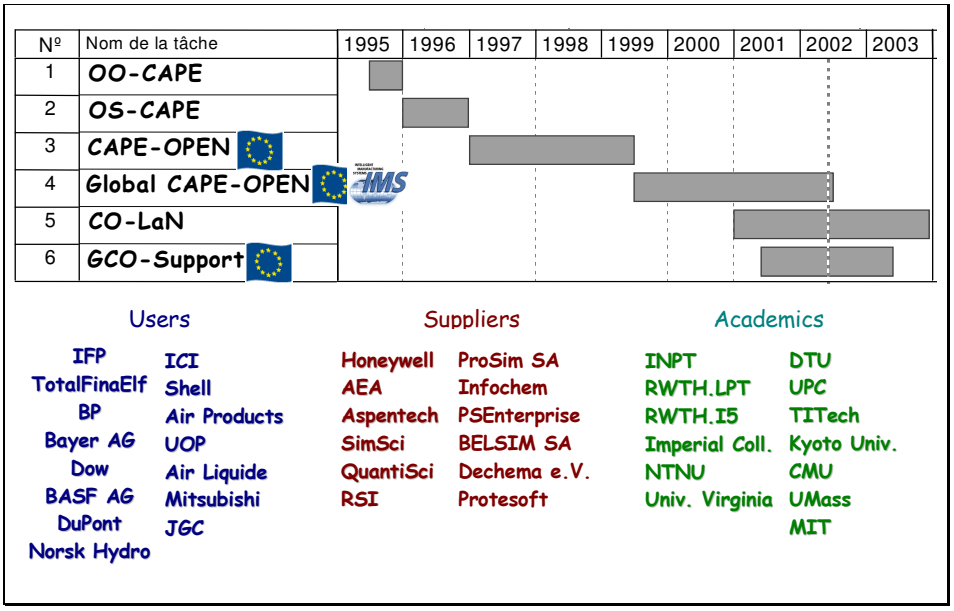


Figure 1 Projects aiming to develop the CAPE-OPEN system

This document is an integrated document aiming at collecting and updating all the former M&T recommendation documents produced by the M&T group for the CAPE-OPEN project and Global Cape-Open project. These former documents are:

- CAPE-OPEN Methods & Tools Guidelines (03_CO_Methods_and_Tools_Recommendations.pdf, September 1999)
- GCO Methods & Tools Guidelines (GCO-MGT-17v1-M&T Guidelines.DOC, January 2000)
- CAPE-OPEN Common Interfaces: Overview (CO Common Interfaces - Overview.doc, v4, October 2001)
- Proposal on a "full" consistent CO CORBA Specification (CO CORBA Specification.doc, v3, September 2000)
- COM Architecture Overview and Basic Principles (COM Architecture Overview.doc, September 2000)
- COM-CORBA Bridging in Global CAPE-OPEN (GCO-MGT-xxv1-COM_CORBA_Bridge.doc, July 2000)
- Update on Types, Interfaces Naming and Undefined Values (Update on types, naming and empty values.doc, v2, June 2001)

This integrated document for the CO-LaN organisation **substitutes now** all these former documents.

2. Foreword

2.1 The M&T group

The role of the Methods and Tools group has been defined in the frame of CO-LaN organisation.

The Methods and Tools group takes care of **assessing, selecting, adapting and applying software methodologies and tools** throughout the CAPE-OPEN (CO) standard lifecycling. Basically, the process defined in CAPE-OPEN, which uses the UML notation with a tailored development process, is followed in development of CO. However, as new methods and net tools will be proposed by the software industry, the M&T group **will continuously monitor the state of the art**, especially in component software and middleware technologies, and provide guidance on how to take advantage of these.

M&T recommendations documents were used at the start of CO initiative.

2.2 M&T Group Membership

Because of the formal role of the M&T group in the approval process for open standard specifications, there is a need to define participation to this group. Therefore the following is proposed. It can be noted, however, that the M&T group does not address consistency between the different interfaces from a CAPE-business view. This is done by the Scientific and Technical Committee.

The initial members of M&T group were before the beginning of CO-LaN organisation: Pascal Roux, Bertrand Braunschweig and Daniel Rahon (IFP), Ben Keeping (Imperial) Jean-Pierre Belaud (INPT), Juan Carlos Rodriguez, Sergio Cebollero and Daniel Piñol (Hyprotech), Lars Von Wedel (RWTH.LPT), Jorg Köller and Alexander Kuckelberg (RWTH.IS), Jacky Bernier (Total), Michael Halloran (Aspentech), Boris Russel (DTU), Robert Huss (UMass). The group leader was Bertrand Braunschweig.

Through the CAPE-OPEN specifications life cycle management mission, the CO-LaN organizes the maintenance, evolution, and expansion of the specifications; this is organized through Special Interest Groups (SIGs), each created to take care of a meaningful subset of the standard. The work done by SIGs follows a well-defined approval process. A specific SIGs is dedicated to M&T topics. The membership of this M&T SIG is not defined at this time.

2.3 A note on tools recommended

Numerous tools are mentioned in this document. Much of the behind-the-scenes work of the Methods and Tools group involves trying out, and thereby informally evaluating these tools.

2.4 Notation and terminology used in this document

2.4.1 Notation

This document uses the MS Word template document (Template for Interface Specification Documents.dot) and so takes advantage of all the associated MS Word styles.

2.4.2 Terminology

- CAPE: Computer-Aided Process Engineering
- CO: CAPE-OPEN, an open standard in Computer-Aided Process Engineering for integration and interoperability of process modelling software components
- COM: Common Object Model (© Microsoft Corporation)
- CORBA: Common Object Request Broker Architecture (© OMG)
- OMG: Object Management Group, a non-profit organization defining interoperability standards
- IDL: Interface Definition Language, a programming language for specifying interfaces
- MIDL: Microsoft Interface Definition Language
- CIDL: (OMG) CORBA Interface Definition Language
- UML: Unified Modelling Language, an OMG standard notation for Object-Oriented software engineering
- OO: Object-Oriented, an approach for software development.
- PMC: Process Modeling Component
- PME: Process Modeling Environment
- M&T: Methods and Tools
- BSCW: Basic Support for Co-operative Work. A web-enabled software providing shared workspaces.

2.5 Where to find additional information

2.5.1 CAPE-OPEN standard specifications

All materials related to CO standard is located on www.colan.org

2.5.2 Other documents

[1] “Inside COM”. Dale Rogerson. Microsoft Press. Washington (USA) 1997.

[2] “The C++ Programming Language” Second Edition. Bjarne Stroustrup. Addison Wesley. USA 1991.

[3] “OLE Automation Programmer´s Reference”. Microsoft Press. Washington (USA) 1996.

[4] “Doing Objects in Microsoft Visual Basic 5.0”. Deborah Kurata. Ziff-Davis Press. Emervylle, California (USA) 1997.

[5] “Understanding ActiveX OLE”. David Chapel. Microsoft Press. Washington (USA) 1996.

[6] “Beginning ATL COM”. Wrox Press Ltd. Birmingham (UK) 1998

[7] www.microsoft.com for COM resources

[8] www.omg.org for CORBA and UML resources

3. Concepts and definitions

3.1 Overview

The *CAPE-OPEN standard* is characterized by a unique and global version number and is described by a set of documents. Each document has its own versioning number in order to track its own life cycling, however it depends on a specific version number. The *CAPE-OPEN formal documentation set* is shown in the Figure 2. It gathers the up-to-date materials - documents, software libraries and software tools - that were produced by the partners along the CAPE-OPEN initiative. Altogether they define the version 1.0.0 of the *CAPE-OPEN standard*.

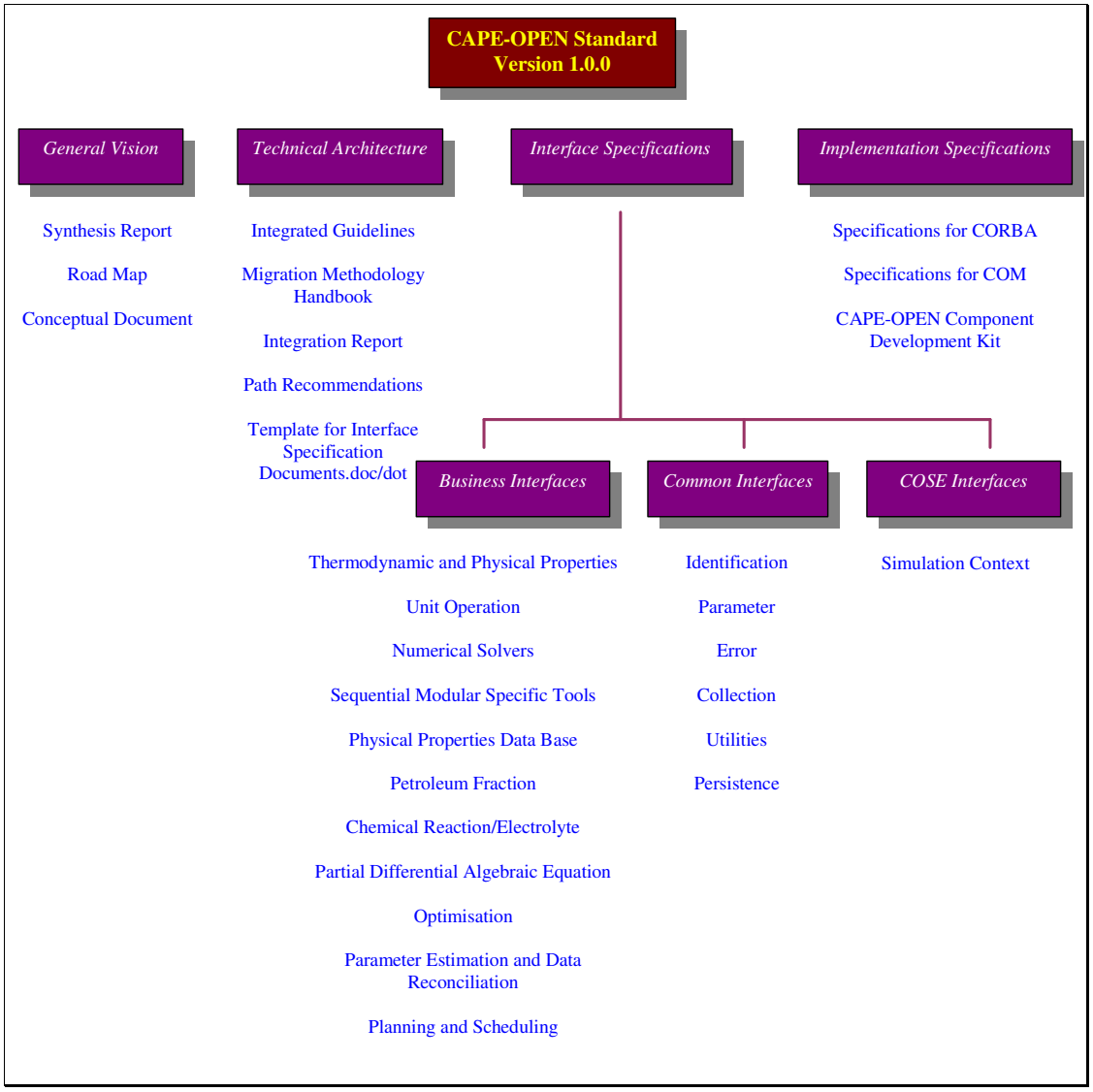


Figure 2 Documentation organisation

The formal documentation set includes the blocks; *general vision, technical architecture, interface specifications* and *implementation specifications*. The third first blocks enclose documents such as pdf files while the latter is formed of software libraries and applications. That is essential to get a well-adapted organisation for managing such complex software standard. The result suits with our software development methodology that respects a kind of model driven style.

The *general vision* folder contains the documents that should be read first for **regular general information** such as general requirements and needs.

The *technical architecture* folder integrates the **horizontal technical materials** and defines an **infrastructure** with the aim of a process simulation based on the *CO standard*. This document belongs to this block.

The *interface specifications* folder encloses a set of **open interface specification document**. Any open interface specification document is available using pdf or HTML file. The *interface specifications* are refined into three inner folders; *business interfaces, common interfaces* and *COSE interfaces*:

- The *business interfaces1* folder owns all **vertical interface specification documents**. The interfaces are **domain-specific interfaces for CAPE application domain** such as Unit Operations, Numerical Solvers, etc. They form the core of conceptual model that leads to CO components which can be involved in an execution of a CO process simulation application.

- *CAPE-OPEN Simulator Executive (COSE) interfaces* refer to **horizontal interface specifications**. They are interfaces for **simulation/modelling environments**. Within this category, services of general use are defined such as diagnostics and material template system in order to be called by any CO components through a call back usage.

- The *common interfaces* folder contains **horizontal interface specification** documents for handling concepts that may be required by any *business interfaces* and *COSE interfaces*. This is a collection of interfaces that **support basic functions** and are always independent of *business/COSE interfaces*.

The *implementation specifications* folder contains the **implementation** of the *interface specifications* - *business interfaces, common interfaces* and *COSE interfaces* - for particular distributed computing platforms. For implementing CO compliant software components any developer has to use these official implementation specifications. The *implementation specifications* are currently available for (D)COM and CORBA platforms through the Interface Definition Language (IDL) libraries.

- *Specification for COM*: Several files expressed in Microsoft IDL integrate all the implementation specifications for COM platform. The corresponding type library - a compiled version of the IDL source - required by the MS-Windows operating system is also available.

- *Specification for CORBA*: One unique file expressed in OMG IDL integrates all the implementation specifications for CORBA platform. No corresponding compiled version is provided because that would be a strong contradiction to the CORBA objective of tools suppliers independence (this point is discussed in section 8).

- *CAPE-OPEN Component Development Kit*: This CAPE-OPEN Component Development Kit (COCDK) gathers a set of software tools in order to help developer in his compliant software component development process. The COCDK encloses software tools that correspond to a certain way in implementing the *CO standard*, so it relies on a specific CO standard version, a given distributed platform and other technical stuff such as implementation languages. This kit focuses on four topics; (i) interworking with the COM-CORBA bridging tool; (ii) rapid application development with the thermo/unit wizards for COM and migration materials, (iii) self-learning with video demonstration and code

[1] Business interfaces can be roughly grouped in four categories; numerics, unit operations, physical properties, and others.

samples; (iv) testing and labelling for COM with the unit, thermo, COSE, SMST, PPDB testers.

The *interface specifications* folder gathers a set of open interface specification document. These key documents contain the description, the explanation and the design of all the CO interfaces. At this level the content of these documents are abstract specifications documents which create and document a conceptual model in an implementation neutral manner. That means that their design is supposed to be clearly independent of any distributed computing platform. So the core of *interface specifications* deals with the conceptual model even if some parts relate to the implementation model for illustration.

3.2 Reference model

In order to make the difference between the service provider and the service requestor, the standard distinguishes **two kinds of CO software components**: Process Modelling Component (*PMC*) and Process Modelling Environment (*PME*), the former providing services to the latter. Typically, the *PMEs* are environments that support the construction of a process model, and allow the end-user to perform a variety of different tasks, such as process simulation or optimisation.

The next figure (using UML notation) identifies and characterises the components, interfaces, and communication protocols. It includes the middleware component that enables the communication in a distributed environment, the CO components (*PMEs* and *PMCs*) and three categories of interfaces (*Common Interface*s, *COSE Interfaces* and *Business Interfaces*).

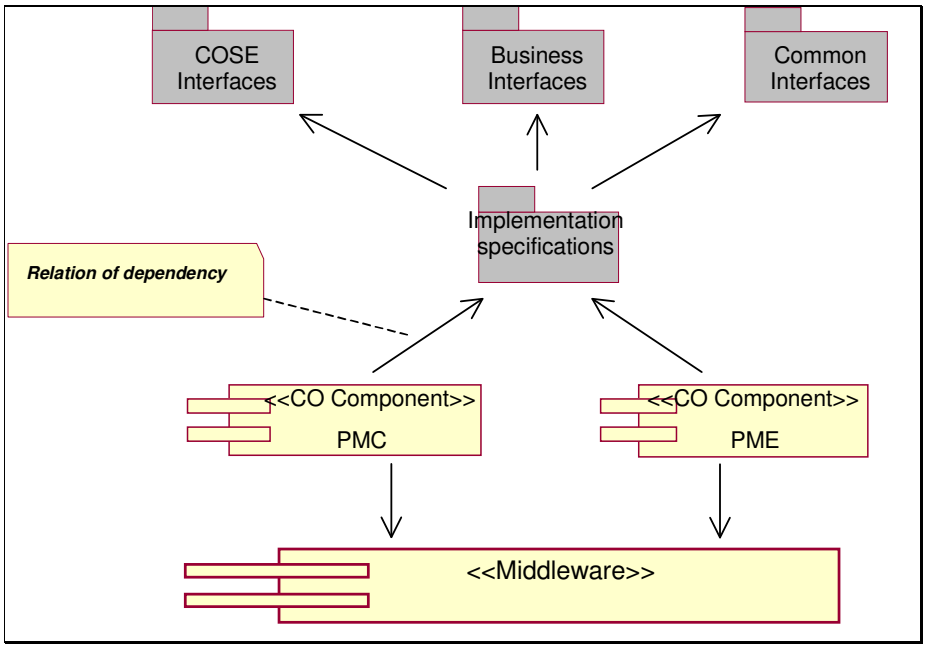


Figure 3 Reference model

The middleware component is the basic mechanism by which objects transparently make requests to and receive responses from each other on the same machine or across a network. It forms the foundation for building open simulation applications constructed from distributed CO components in both homogeneous and heterogeneous environments.

3.3 CO object

An object is an instance of a class. An object satisfies a CO interface if it can be specified as the target object in each potential request described by the CO interface. It belongs to the implementation step and so is out of the CO scope. However this proprietary object is specific from the other usual proprietary objects since it collects the CO calls so, we call it a *CO* (interface) *object*.

It is worth noting that we can have not only *CO objects* within a *PMC* but also *CO objects* within a *PME*. For instance obviously there are such objects in a *PMC* unit operation respecting the *Business Interface:* Unit Operation interface specification. But there can be also such objects in a simulation environment *PME* respecting the *COSE Interface:* diagnostic interface specification. Nevertheless *CO objects* coming from *PMC* can be viewed in a different way from *CO objects* coming from *PME*. As a matter a fact from an architectural point of view the *PME objects* are currently only designed to be called by *PMC objects* with respect to a call-back pattern.

4. Selected Methods and Tools

4.1 Main technical choices

The following technical decisions are supported by work done during the CO initiative and by individual achievements of many partners involved in the development of modern software for process simulation.

(i) The standard interfaces should be defined and expressed using an **object-oriented approach**. The OO approach is currently the best technical solution for developing interface standards. It also encompasses the "conventional" procedural approach.

(ii) The standard interfaces assume that a process simulator is made of several **components**.

(iii) The standard interfaces should use existing **middleware**, namely **ActiveX/COM** and **CORBA**. More specifically, the standard interfaces should be expressed **in both forms** in order to be future-proof.

(iv) The standard interfaces should be applicable to **several hardware platforms and operating systems**.

(v) There should be a distinction between **project work** and **the standard**. The life-span of the standard is expected to be much greater than the duration of the initial CAPE-OPEN project itself. Choices made for the project - because it has limited resources and duration - should not compromise the quality and scope of the standard.

(vi) The standard interfaces should allow the encapsulation of **legacy code**.

4.2 Notation and Interface Definition Languages

4.2.1 Notation

We adopt the **Unified Modelling Language (UML)** for the CAPE-OPEN set of object models. UML unifies the popular OMT, Booch and Jacobson methods for object-oriented projects, and it is becoming a de facto standard with high acceptance from the OMG. UML is seen as the way forward for CAPE-OPEN.

4.2.2 Implementation specifications

We express the interface specifications both for ActiveX/COM and CORBA. This means that in addition to the formal UML models coming from analysis the CAPE-OPEN interfaces are expected to be expressed in two parts, one describing the COM specification, one describing the CORBA specification.

The COM specifications are expressed in MIDL, the **Microsoft Interface Definitions Language**.

The CORBA specifications are expressed in IDL, **OMG's Interface Definition Language**.

We will evaluate bridge capabilities between COM and CORBA.

4.3 Modelling tools

We do not recommend a single modelling tool for developing the object models. As a consequence, the UML models will be prepared with different tools (Rational Rose, Select, P+, Visio etc.) until we can recommend the use of a single one. As another consequence, the COM and CORBA specifications will not systematically be automatically generated from the UML models.

4.4 Work process

The definition of CO interfaces is done following a development process based on the UML object-oriented notation for all formal models of the interfaces, including the user requirements, producing use cases, sequence diagrams, state transition diagrams, class diagrams and, finally, interface diagrams which accompany the corresponding middleware implementation.

The work process that we follow for each sub-task of CAPE-OPEN technical work expected to deliver standard interface specifications and prototypes is presented in next Table.

In practice, an iterative approach where the different models and implementations were subject to progressive refinements had to be adopted. Overall, this work process proved to be both an efficient and an effective mechanism for developing commonly agreed standard interface specifications and prototypes meeting those specifications, in an initiative involving a relatively large number of actors with widely different backgrounds.

Phase	**Step**	**Goal**
REQUIREMENTS	User requirements, text	Requirements in textual format
REQUIREMENTS	User requirements, Use-Cases	Use Case model
DESIGN	Design Models	Sequence, state transition, and interface models
SPECIFICATION	CO-COM Implementation Specification	Draft interface specification in Microsoft IDL
SPECIFICATION	CO-CORBA Implementation Specification	Draft interface specification in OMG IDL
IMPLEMENTATION	COM Implementation	COM prototype implementation
IMPLEMENTATION	CORBA Implementation	CORBA prototype implementation
VALIDATION	Standalone Testing	Tested component
VALIDATION	Integration testing	Tested specification
DELIVERY	Open interface specification document	Approved specification. MS-Word/pdf document with IDLs files and prototypes binary code

Figure 4 Work process phases

4.4.1 Requirements

The goal of the Requirements phase is to produce a structured set of users requirements in the form of a textual description and of a set of Use Cases.

TEXTUAL REQUIREMENTS

Express the user requirements in written form. This should be obtained by consensus of the interface designers team. No tools are used at this stage. The result is an MS-Word document which presents and justifies the requirements. For the *Petroleum Fractions Interface Specification*, an example could read as follows:

> …
>
> The CAPE-OPEN material template contains the component (species) information. To handle petroleum assay and fraction, the material template should be extended to include assay and fractions. To allow tracking of petroleum properties as the petroleum fractions go through the flowsheet from one Units model to another (and may change in the Reactor model), the material template should be extended to include the petroleum properties as attributes.
>
> …

USE CASES

Build Use Case Models from the Users Requirements, using the UML notation. The Use Case models express the core requirements and provide a basis for testing a proposed design. These models should be obtained by consensus of the interface designers team. No tools are required (i.e. the drawing capacities of MS-Word suffice) although the use of UML diagramming tools is encouraged. The result is an MS-Word document including the models. The following example is the Use Case map from *Petroleum Fractions Interface Specification*:

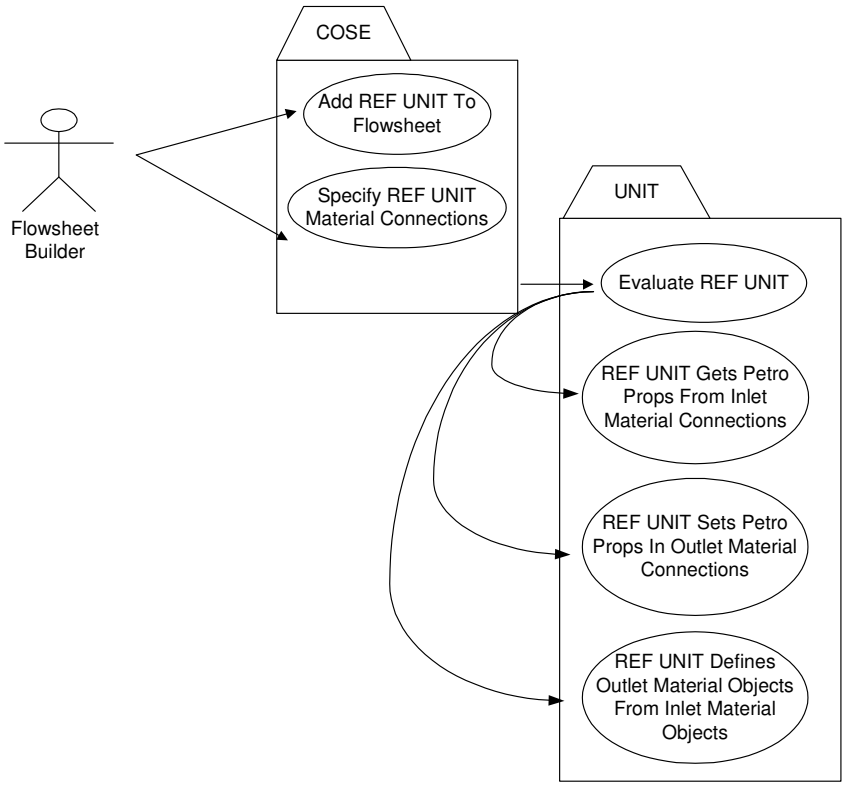


Figure 5 Use Case map example

As example the following is the description of the [Add REF UNIT To Flowsheet] use-case:

Actors: Flowsheet Builder

Priority: As in UNIT Use Case Add UNIT to Flowsheet

Classification:

Context:

Pre-conditions: As in UNIT Use Case Add UNIT to Flowsheet

Flow of events:

As in UNIT Use Case Add UNIT to Flowsheet, but the UNIT is a Refinery UNIT.

And:

The Simulator Executive asks the UNIT is it is a Refinery UNIT (i.e. requires petroleum fraction properties)

If so, the Simulator Executive will ask the UNIT if it requires re-characterization of petroleum fraction properties (see note below).

Note:

Typically, a Simulator Executive not aware of "component continuous properties" (SIM_A) will need duplicating the component slate in the outlets to "emulate" continuous properties, while in a Simulator Executive able to cope with "component continuous properties" (SIM_B) this is not required.

In both cases (SIM_A and SIM_B) the UNIT will be presented with a single component slate, and therefore, for a UNIT there will be no difference regarding the capabilities of the Simulator Executive.

SIM_A will have a set of pseudo components in the inlets (e.g. HYP_GROUP_1) and two sets of pseudo components in the outlets (e.g. HYP_GROUP_1 and HYP_GROUP_2). Flow rates of HYP_GROUP_1 in the outlets will be all zero, since UNIT results will be in fact represented in HYP_GROUP_2.

To emulate continuous properties (from the perspective of a UNIT), SIM_A will have to clone outlets, and remove the additional component slate HYP_GROUP_2. The cloned outlet will be the one presented to the UNIT (i.e. contain a single component slate, e.g. HYP_GROUP)

When the UNIT operates on outlet hypo group (HYP_GROUP), SIM_A will know these operations have to actually be performed on HYP_GROUP_2 and not on HYP_GROUP_1.

Post-conditions:

As in UNIT Use Case Add UNIT to Flowsheet.

Errors:

As in UNIT Use Case Add UNIT to Flowsheet

Uses:

Extends: UNIT Use Case <Add UNIT to Flowsheet>

4.4.2 Analysis and design

The goal of the Analysis and Design phase is to produce a set of design models using the UML notation suitable for the requirements expressed in the Requirements phase. This model tries to be the result of an analysis and design phase independently of any middleware implementation.

Build Sequence, Interface, Component models following the UML notation. Also each interface is described precisely. These models are essential for the design of interface objects and are used in a later stage for preparing the interface specifications. No tools are required (i.e. the drawing capacities of MS-Word suffice)

although the use of UML diagramming tools is encouraged. The result is an MS-Word document including the models. The following examples come from *Petroleum Fractions Interface Specification*:

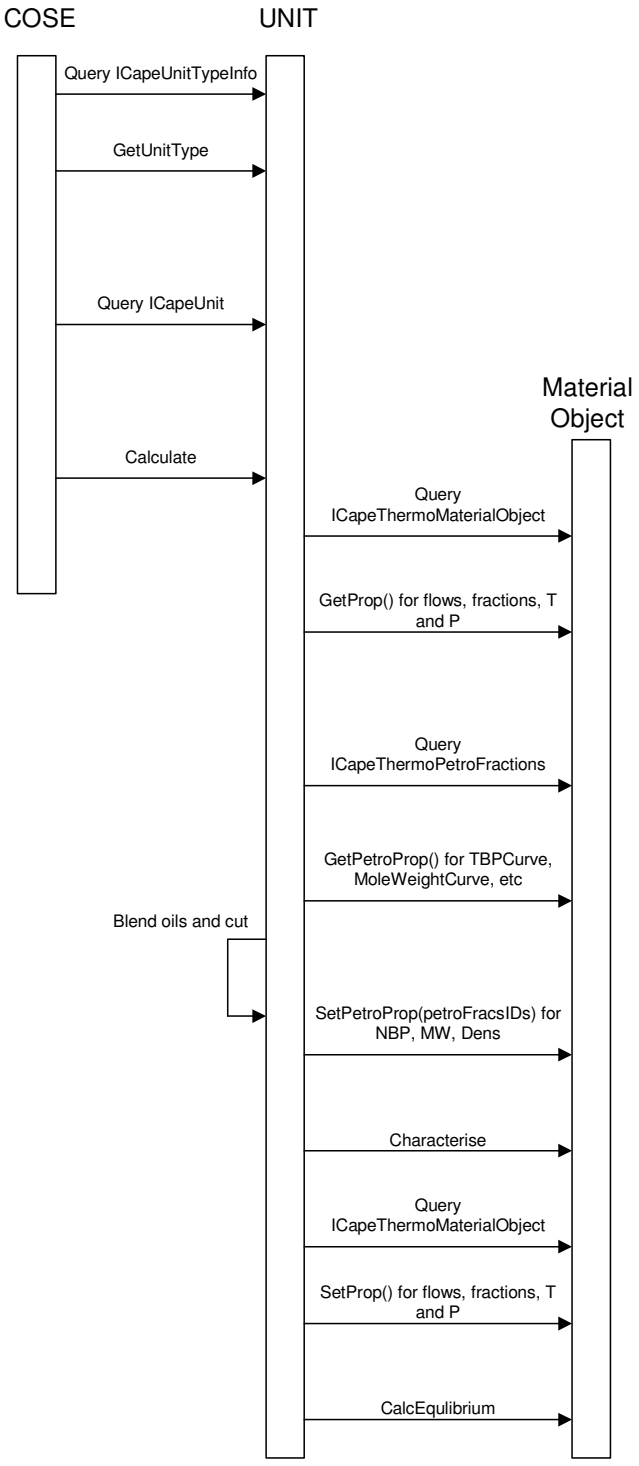


Figure 6 Sequence diagram example

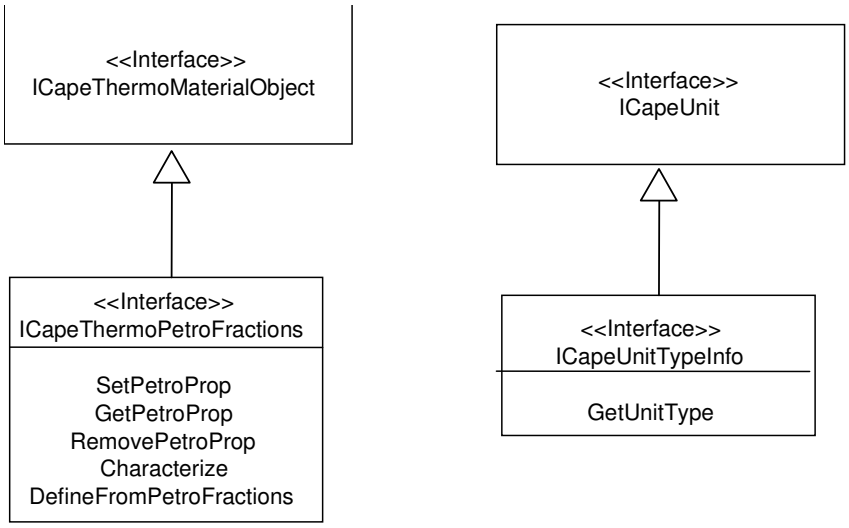


Figure 7 Interface diagram example

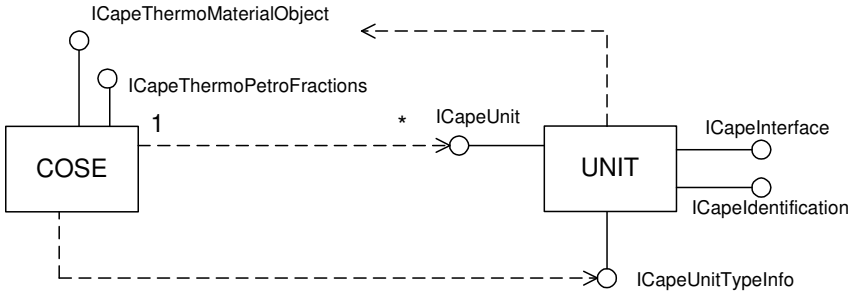


Figure 8 Component diagram example

4.4.3 Interface specifications

The goal of the Specifications phase is to produce the CAPE-OPEN Interface Specifications in Microsoft IDL and OMG IDL from the design models established in the Analysis and Design phase. Some notes can give clarifications and details on this "translation" from the middleware-independent design to middleware-dependant specification.

INTERFACE SPECIFICATION IN MICROSOFT IDL

Microsoft IDL specifications for interface objects are built from the design models developed in the design phase.

No tools are recommended at this stage: a text editor suffices. However, automated generation of some of the Microsoft IDL specifications could be envisaged later on during the project, depending on the reassessment of UML compatible tools.

INTERFACE SPECIFICATION IN OMG IDL

IDL specifications for interface objects are built from the design models developed in the design phase.

No tools are recommended at this stage; a text editor suffices. However, automated generation of some of the IDL specifications could be envisaged later on during the project, depending on the reassessment of UML compatible tools.

4.4.4 Prototypes Implementation

The goal of the implementation phase is to produce prototype COM and CORBA compliant components using the interface specifications developed in the Specifications phase. Many of these components encapsulate legacy code.

COM PROTOTYPE AND WRAPPING OF LEGACY CODES

Prototype COM-compliant implementations are produced by generating the interface code with the Microsoft IDL (MIDL) compiler and encapsulating other pieces of code within the interface. The generated interface code comprises:

(i) a set of header files defining the interfaces;

(ii) code for proxies and stubs needed for local and remote methods invocation;

(iii) code for filling the COM object library.

Legacy code in FORTRAN or other language has to be wrapped.

The use of the **MIDL compiler** is mandatory for this phase.

The result of this phase is a binary executable expected to work with ActiveX/COM and ready to be tested.

CORBA PROTOTYPE AND WRAPPING OF LEGACY CODES

Prototype CORBA-compliant implementations are produced by generating the interface code with an IDL compiler and encapsulating other pieces of code within the interface. The generation of interface code comprises

(i) class declarations for the interface

(ii) additional code for client-server communication

(iii) skeletons to be used for the implementation of the interface

Legacy code in FORTRAN or other language has to be wrapped. The use of an IDL compiler is mandatory for this phase. The result of this phase is a binary executable expected to work with any CORBA 2 or upper ORB (Object Request Broker) and ready to be tested. No specific ORB product is recommended knowing that CORBA 2.0 and upper allows inter-ORB communication.

4.4.5 Validation

The goal of the Validation phase is to do a quick standalone test of the prototype components before submitting them to the integration validation. The tests are expected to be done by the developers and should not need external review unless specific problems arise.

The result of this phase is a set of tested prototype components ready to be delivered for integration in the validation environment. Both COM and CORBA implementations need such validation.

5. Software engineering elements

This part provides software elements for developing CAPE-OPEN interface specifications.

The areas considered in this part are as follows, together with the section number:

(i) Data Types to be used in arguments of interface methods;

(ii) Undefined values when a value is not available;

(iii) Arrays of object or elementary types:

(iv) Naming : how names are given to interfaces, methods, properties and arguments;

(v) Argument Ordering : order of arguments in interface methods;

5.1 Elementary types

The CAPE-OPEN standard is adopting a standard set of data types that are handled by the CAPE-OPEN interfaces. These types are independent of the component implementation language, and middleware, but they must be capable of being easily mapped to the middleware or implementation language types that are used. Developers of prototypes define the mappings from these CAPE-OPEN types to middleware data types.

The most common set of data types to appear in interfaces are presented in the table below. T refers to any other CAPE-OPEN type, e.g. an object (ICapeUnit) or a basic type such as Long.

CAPE-OPEN	**Analysis**	**COM**	**CORBA**
CapeLong	long	long	long
CapeShort	short	short	short
CapeDouble	double	double	double
CapeFloat	float	float	float
CapeBoolean	boolean	VARIANT_BOOL	boolean
CapeChar	char	BYTE	char
CapeString	string	BSTR	string
CapeDate	string date	DATE	string
CapeURL	URL string	BSTR	string
CapeVariant	container of any other type	VARIANT	any
CapeInterface	CO interface	LPDISPATCH	Object
CapeArrayT	array of T	VARIANT	sequence<T>

The defined CAPE–OPEN data types are used in IDL files with appropriate definitions for the specific middleware types. The CapeInterface type is used when passing other interfaces through argument lists of methods. So for example in Unit we may have a method on the ICapeUnit interface named GetPorts, which could return the ICapeUnitPort interface and then this would be represented generically in IDL by (however in some cases CO interface designers could make use of strong typing in the case of CORBA IDL):

```
interface ICapeUnit
{
...
GetPorts([return] CapeInterface portsInterface)
...
}
```

It may be useful to add some missing types among the above elementary types. For example we could add:

```
long long     CapeLongLong
long double   CapeLongDouble
wchar         CapeWChar
wstring       CapeWString
```

On the CORBA side, as these types are not available for all platform we choose not to add them at this time.

5.2 Undefined values

In some CO specifications such as Physical Properties Data Base, information can be unavailable and that is not an abnormal process (CO error handling should not be applied). Therefore as a common specification for all interfaces the following undefined constants are defined:

- CORBA part

```
const  CapeLong      CapeLongUNDEFINED   =-2^31;
const  CapeShort     CapeShortUNDEFINED  =-2^15;

const  CapeDouble    CapeDoubleUNDEFINED =-4.9E-324;
const  CapeFloat     CapeFloatUNDEFINED  =-1.4E-45;

const  CapeChar      CapeCharUNDEFINED   ='\0';
const  CapeString    CapeStringUNDEFINED ="";
const  CapeDate      CapeDateUNDEFINED   ="";
const  CapeURL       CapeURLUNDEFINED    ="";

#define CapeArrayLongUNDEFINED          NULL;
#define CapeArrayShortUNDEFINED         NULL;

#define CapeArrayDoubleUNDEFINED        NULL;
#define CapeArrayFloatUNDEFINED         NULL;

#define CapeArrayCharUNDEFINED          NULL;
#define CapeArrayStringUNDEFINED        NULL;
#define CapeArrayDateUNDEFINED          NULL;
#define CapeArrayURLUNDEFINED           NULL;
```

- COM part

```
// Commented out because it causes MIDL to generate a .h file that
// does not compile, needs further investigation.
//
//const       CapeLong            CapeLongUNDEFINED   =-2^31;
//const       CapeShort           CapeShortUNDEFINED  =-2^15;
//
//const       CapeDouble          CapeDoubleUNDEFINED =NaN; [2]
```

[2] NaN: Not a Number, defined by the IEEE double float format, e.g. 7fff ffff ffff ffff (0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111), Test for NaN of floating point number x: ((x & 7ff0 0000 0000 0000)>0)&&((x & 800f ffff ffff ffff)>0) returns true, if x is not a number; false otherwise. & is the bitwise AND, && the logical AND. See also function “isnan” of library libm.so of Solaris systems or equivalent libraries.

```
//const      CapeFloat           CapeFloatUNDEFINED  =NaN; [3]
//
//const      CapeChar            CapeCharUNDEFINED   ='\0';
//const      CapeString          CapeStringUNDEFINED =NULL;
//const      CapeDate            CapeDateUNDEFINED   =NULL;
//const      CapeURL             CapeURLUNDEFINED    =NULL;
```

Notes on COM implementation of undefined values for strings and variants types:

BSTR types : CapeString and CapeURL

- C++

```
//Valid Implementation:
BSTR strArg = NULL;

//Invalid implementation:
BSTR strArg = ::SysAllocString(L"");
```

- VB

```
// Valid Implementation:
Dim strArg as String
strArg = vbNullString

// Invalid implementation:
Dim strArg as String
strArg = ""
```

VARIANT:

- C++

```
// Valid Implementation:
// the vt type of the VARIANT is
// set to VT_EMPTY
VARIANT VarArg;
VariantInit(&VarArg);

// Invalid implementation:
VARIANT VarArg;
```

- VB

```
// Valid Implementation:
Dim VarArg as Variant
VarArg = Empty

// Invalid implementation:
Dim VarArg as Variant
VarArg = vbEmpty
```

[3] NaN: Not a Number, defined by the IEEE single float format, e.g. 7fffffff (0111 1111 1000 000 0000 0000 0000 0000), Test for NaN of floating point number x: ((x & 78000000)>0)&&((x & 87ffffff)>0) returns true, if x is not a number; false otherwise. & is the bitwise AND, && the logical AND. See also function “isnan” of library libm.so of Solaris systems or equivalent libraries.

5.3 Arrays

It was decided that a common naming has to be used. Therefore the term for the list of something is the same for the analysis view and the implementation specification view (COM and CORBA).

For CORBA we have:

```
typedef sequence<Type> CapeArrayType.
```

Note that we always use the sequence CORBA type and not the array CORBA type. Therefore we get for the elementary types such arrays:

- CORBA

```
typedef sequence<CapeLong>      CapeArrayLong;
typedef sequence<CapeShort>     CapeArrayShort;
typedef sequence<CapeDouble>    CapeArrayDouble;
typedef sequence<CapeFloat>     CapeArrayFloat;
typedef sequence<CapeChar>      CapeArrayChar;
typedef sequence<CapeString>    CapeArrayString;
typedef sequence<CapeBoolean>   CapeArrayBoolean;
typedef sequence<CapeDate>      CapeArrayDate;
typedef sequence<CapeURL>        CapeArrayURL;
typedef sequence<CapeVariant>   CapeArrayVariant;
typedef sequence<CapeInterface> CapeArrayInterface;
```

- COM

```
typedef VARIANT           CapeArrayLong;
typedef VARIANT           CapeArrayShort;
typedef VARIANT           CapeArrayDouble;
typedef VARIANT           CapeArrayFloat;
typedef VARIANT           CapeArrayChar;
typedef VARIANT           CapeArrayString;
typedef VARIANT           CapeArrayBoolean;
typedef VARIANT           CapeArrayDate;
typedef VARIANT           CapeArrayURL;
typedef VARIANT           CapeArrayVariant;
typedef VARIANT           CapeArrayInterface;
```

5.4 Naming

For CAPE-OPEN interface specifications, the following guide is used in order to provide naming conventions for the actual interfaces, for the methods which belong to interfaces, and for the arguments which belong to methods.

English is the base language to be used. Names should contain a clear indication of the purpose of the interface, method or argument and use mixed case words not allowing the underscore character (_) as a separator. There is no maximum length for names.

Care should be taken in the use of abbreviated words. The meaning of shortened words may not be obvious to non-native English speakers, or to people that are not involved in the software development process.

In naming interfaces, it should be noted that Microsoft IDL and CORBA IDL are case sensitive. However, CORBA IDL does not allow different names that differ only in case.

In this document the IDL used by Microsoft and the IDL used by OMG are frequently encountered. They are referred to as MIDL and CIDL in order to simplify the text of this document. When the term IDL is used alone, it implies a generic CAPE-OPEN interface language (IDL), rather than a particular implementation.

5.4.1 Interfaces

The CAPE-OPEN interfaces themselves are prefixed as:

ICapeTopicName

where *I* implies interface, *Cape* refers to the domain, and *Topic* is the scope of the common/business/COSE interface such as Unit, Thermo, Numeric, Parameter, etc… Following this prefix is the function *name* such as Solver, Port, MaterialObject, PetroFractions, …

As an example, for the numerical scope we have an interface named

ICapeNumericLinearSolver

And following the same rules, for the Unit, we might have:

ICapeUnitPort

5.4.2 Attributes and Methods

Within an interface there are methods to perform defined services of the interface. CO methods should start with an upper case. These methods can be classified as follows:

(i) Properties/attributes access methods

(ii) Object creation and object release and/or destruction

(iii) Object enhancement or extension

(iv) Specialist domain behaviors (such as pre-conditioning an iterative solver)

Methods used to provide standard base services of the CAPE-OPEN interfaces should use names that map naturally to their middleware counterparts if this is possible. For each of these groups, the methods names should follow a standard convention to aid someone who wishes to use the methods in the interface or in the future to extend the interface. The proposed naming convention for each of these is as follows:

Type of method	**Prefix**
Data obtaining	Get
Data providing	Set
Data inquiring	Query
Object creation	Create
Object release	Release
Object destruction	Destroy
Object enhancement	Add
Object restrict	Remove
Object duplication	Clone
Specialist behaviour	[none]

Figure 9 Prefix of method name

Object release is used in COM and in CORBA it is supported as well, but not across address spaces. Object destroy is needed on the CORBA side since CORBA does not provide any other method to shut down an

object by releasing it from the client. We have identified the need for object duplication, therefore we have defined a method of name «Clone».

5.4.3 Arguments

Interface method arguments should be named so that they define clearly their purpose. In the full IDL produced, the type of the argument is also needed. This type information must reflect upon the types CAPE-OPEN supports in its interfaces as described previously. Attributes should begin with a lower case letter and multiple words should capitalise the initial letter of the second and subsequent words.

Examples are:

pressure, deltaPressure, numTraysInColumn

5.5 Argument ordering

In methods within interfaces, the arguments to those methods should always have a consistent ordering to aid clear reading from the user. Read arguments should appear first and be marked with the [in] IDL attribute, followed by read-write arguments marked with the [in,out] attribute, then write-only arguments marked with the [out] attribute. Finally in MIDL the function return value marked with the [out,retval] attribute whereas in CIDL it is implied by the return type. For the generic IDL the return value should be marked as [return].

- So in MIDL we would have

```
HRESULT Methodname ([in] type readarg1,…, [in,out] type rwarg1,…, [out] type warg1,…,
[out,retval] type* retval)
```

Due to VB issues, [out] is not used. [in,out] is used using the following define statement:

```
#define ACTUALLYout in,out
```

- In CIDL we would have

```
type Methodname (in type readarg1, ..., inout type rwarg1, ...,out type warg1,...);
```

and for methods that may raise exceptions:

```
type Methodname (in type readarg1, ..., inout type rwarg1, ...,out type warg1,...) raises
(Exception1, ...);
```

6. General architectural and technical Issues

6.1 Architectural aspects of interfaces

Each interface specification document produces interface specifications that include the interface diagram and a generic UML model. This keeps the CAPE-OPEN interfaces independent of the middleware implementation in MIDL or CIDL. The specification releases also provide both MIDL and CIDL because both COM and CORBA prototypes are being implemented.

The following is given as a guide to the production of specifications for the MIDL and CIDL versions. The objective of this approach is to use the different strengths of COM and CORBA implementations in the CAPE-OPEN prototyping work.

6.1.1 COM

The analysis version of the interface diagram is prepared showing inheritance with a traditional inheritance notation. For the MIDL specification, the interface designers should write the MIDL assuming an implementation that is not done with custom interface inheritance. The actual design/implementation work can then decide whether to handle such inheritance with delegation/aggregation, containment techniques or whether to use some other alternatives.

All COM interfaces are dual interfaces, directly inherited from IDispatch. This is a common recommended COM practice that allows users of scripting languages, such as Visual Basic for Applications (VBA), to access properties and methods.

6.1.2 CORBA

The analysis version of the interface diagram also shows an inheritance relationship with the traditional UML notation. However, in this instance, the interface diagram can map the inheritance relationships directly into the CIDL.

6.2 Advices on interfaces design

6.2.1 Factoring of interfaces in COM

One of the biggest issues to be considered when designing interfaces is **factoring**. Factoring is the process by which you decide how many interfaces to design, how many methods each of the interfaces have, and how many parameters each of the methods has. An entire book could be written on strategies for factoring interfaces, and there is much literature available on the topic of object-oriented analysis and design that is applicable. However, there are some basic rules you can use as you design your interfaces. These rules are described in the following sections.

NUMBER OF METHODS PER INTERFACE

Experience has shown that interfaces with fewer methods are better. Interfaces with many methods that are intended to be implemented by a large number of objects usually end up having most of the methods return E_NOTIMPL.

Fewer methods, however, means more interfaces. The greater the number of interfaces, the greater the number of times a client might be forced to call QueryInterface just to execute a simple task. The general rule is if two sets of functions are independent, that is, you expect either to be implemented without the

other, the sets of functions should be contained in different interfaces. In most cases, if you are tempted to have a "capability flag" to indicate whether some functions are implemented, you should separate interfaces and take advantage of QueryInterface instead.

Also, try to eliminate options no one will want to use or implement. Often, interface designers try to think up every conceivable use for their interface and thus add additional methods to satisfy these "potential" users. Do not fall into this trap. Instead, focus on your primary users and design the interface so that it fits their needs. If a customer needs additional, special functionality, you can provide that functionality in another interface.

NUMBER OF PARAMETERS PER METHOD

When factoring your design, think about "round trips". Each call to an interface method involves at least one "round trip", potentially across a process or machine boundary. Therefore, it is "cheaper" to send everything needed to execute a call with one method than to have to call two methods with half as many parameters. However, it is sometimes possible to reduce the amount of data marshalled by doing just the opposite: have one "setup" method and then let users call the various "worker" methods without having to supply the "setup" information each time.

Also, try to limit the number of parameters a method contains. Having to call a method that takes more than five or six parameters is bothersome to many programmers (and you may start reaching the bounds of what the programmer's compiler can handle).

6.2.2 Note on preparing UML interface diagrams

Interface diagrams used by CAPE-OPEN are an extension of the official UML. However, they are consistent with the thrust of UML and have been found to be a useful tool for moving from use cases to interface specifications. They are required as part of the interface specification documentation. The following comments are provided for guidance.

The aggregation relation within the interface diagrams seemed to confuse some members. The intention behind drawing an interface aggregation is not to specify in MIDL an interface that consists of some other interface, but to show the actual aggregation relationship between the objects that are modeled. MIDL definitions will not use aggregation relationships or custom interface inheritance implicitly, whereas the CIDL can.

6.3 Running components in-process, out-of-process (local and remote)

There should be no implications for the design of the interfaces at the design stage as regards the memory space in which the component implementing the interface should run. It should be noted here that in-process server components are much more efficient than out-of process servers. Thus, whenever the overhead imposed by the calling represents a high percentage of the overall calculation time (e.g. in a simple Unit Operation, or for flashing a Material), an in-process server would be preferred.

6.4 Registry

The middleware environment provides registration. In CORBA this feature is implemented with the Naming and Trader Services which may be used. COM/ActiveX uses the Windows Registry.

There is a naming hierarchy of CAPE-OPEN components. The CAPE-OPEN interfaces are logically grouped in categories that are shown in next COM section.

This classification of the components into categories assists applications which can offer to users only those external components which claim to be of the right category. Obviously, further checks is made to establish

that the components do actually support the CAPE-OPEN standard interfaces. Note that this categorisation does not imply that any extra interfaces need to be developed and that also components can be in more than one category (or subcategory).

6.5 Errors

There are different mechanisms to handle errors within both middleware architectures. COM uses HRESULT return types, whereas CORBA uses an exception mechanism similar to C++/Java. For CAPE-OPEN at least two types of errors should be distinguished:

- **Errors**: Report error when the contract between caller and callee was violated in a manner that the calculation could not be finished. Those errors require user interaction or notification.

- **Warning**: Report warning when the calculation is performed, but some problem occurs that may influence the result and should be reported to the user, e.g. violating the valid range for some physical property correlation.

The authors of interfaces define the contracts for each methods and property access. A violation of the contract, such as passing a Word document object to a physical property calculation, should result in an exception or the return of an error HRESULT, as appropriate for the platform. An exception or error return would typically result in a break in the program flow in the calling routine. Handling an error or exception is typically much more expensive than a successful function return so methods should be defined to avoid using exceptions and error result to communicate conditions that can do not justify a break in program flow. For example, if it is anticipated that the caller would be able to take corrective action for an extrapolation beyond the recommended range for a correlation, extrapolation should be communicated by a different mechanism. The documentation of each method should list anticipated error codes or exceptions.

The *Common Error Interface* deals with the error topic. It explains how to manage in CO standard the error situation using the COM and CORBA architectures.

6.6 Version control of interface specification document

The following practice is used for versioning a specific open interface specification document. It is not relevant to the versioning of CO standard. Version numbers consist of three digits separated by dots, e.g. 0.5.8. The version numbers are increased as follows:

- last digit: the last digit is increased according to the development releases by the organisation responsible for the prototype and for describing the actual interface implementation. For a release, such as 0.5.8, the particular organisation will use its own release scheme for the actual software, e.g. Version 0.5.8 Build 3.

- middle digit: this version number is the responsibility of the interface designers team leader. Each new release on this level is given to validation, in the beginning it is stand-alone validation, towards the end of the project, and it is integration testing.

- first digit: increasing the first digit is in the responsibility of the CO-LaN organisation, and it is assigned to the project leader. That means, it cannot be changed, unless integration testing was successful.

When a release is published, it must include the following as a minimum:

(i) UML descriptions (sequence diagrams, interface diagrams, etc.)

(ii) Documentation (e.g. help files). The documentation could be provided in HTML format and includes the generic CAPE-OPEN IDL form of the interfaces.

(iii) CIDL/MIDL definitions for COM AND CORBA.

(iv) Binary of any simple component prototype implementing the interface

(v) Source code for a simple test harness or other supporting code

All these items should be clearly identified with the version number of the release to ensure consistency. The documentation should mention the releases of other prototypes that the release is designed to operate with for backward compatibility.

The CO standard versioning system is not “directly” related to the version numbers of interface specification documents.

6.7 COM-CORBA bridging

This part located in annexe describes the results of the COM-CORBA bridging activities carries out in the first half of 2000. It describes the bridging prototype based on the IK-CAPE thermo package which was developed to create a real world example for the integration of different middleware approaches within the project.

7. COM-specific architectural and technical issues

This part describes the general principles of COM. The COM Object Model, the most important standard interfaces of COM and some general rules for implementers of COM Components.

This part is not intended as a comprehensive guide of COM, but as a general introduction to the technology, in order that CAPE-OPEN Component developers can get a grasp of what COM is about. It also provides a practical guide for developing and deploying CAPE-OPEN components. The guide is focused on using the VB language due to its simplicity.

7.1 Introduction

As Visual Basic and other scripting languages, COM only supports interface inheritance. Therefore, rather than providing mechanisms for reusing code through direct code inheritance, COM objects reuse code by other techniques involving co-operation of several objects (see aggregation or containment, later in this chapter).

COM (Component Object Model) is a Microsoft standard that establishes rules for implementing components that can be dynamically interchanged and linked to a particular application. COM is not a programming language but a binary standard for connecting components. COM deals with interfaces between components rather than with the components themselves.

There is no guarantee that a compiled C++ component will work when trying to link it together with other objects or applications that were generated with a different compiler. The goal of COM is to provide binary compatibility between components that need to be distributed out of the originating organisation.

This part describes the main concepts of the COM architecture, as well as providing a practical guide for developing and deploying CAPE-OPEN components. The guide is focused on using the VB language due to its simplicity. The practical guide is also intended to aid developers (with experience in programming COM components with other languages) on how to apply their COM skills on developing CAPE-OPEN components.

7.2 COM Interfaces

In the following paragraphs we will be describing some of the most remarkable concepts that make COM a widely used binary standard. In some particular points fragments of code are included with the aim of illustrating those concepts, but you encouraged to read some of the titles included in the list of references, where you will find a more detailed and exhaustive information.

7.2.1 vtbl functions, the foundation of COM interfaces.

For a C++ programmer an interface can be seen as a pure abstract class that, when compiled, gives a specific memory structure consisting of a virtual function table (e.g. an array of pointers to the different functions contained in the class) and a pointer to the virtual table (vtbl). Every component inheriting from this base class will inherit same memory structure.

A component can implement more than one interface (e.g. in the C++ analogy it can multiple inherit from more than one pure abstract base class). If this is the case, the vtbl will be expanded to accommodate the new member functions of other interfaces, and the memory layout will also contain necessary pointers to specific vtbl locations that represent the beginning of every interface.

Interfaces represent how a client communicates with the component, and vtbl pointers represent the entry points for accessing the functionality those interfaces offer.

7.2.2 IUnknown (COM interfaces)

Nevertheless, for a component to be a COM Component the above memory layout is not enough, and the vtbl of its interfaces has to contain the addresses of three additional functions (i.e. QueryInterface, AddRef and Release).

These three methods are the behaviors of a crucial COM interface called IUnknown and therefore, it follows that every COM interface has to inherit non-virtually from IUnknown. This is the way every COM interface can be treated polymorphically.

QUERYINTERFACE

QueryInterface is the key member function of IUnknown and therefore, of COM interfaces. The purpose of QueryInterface is allowing client components to navigate through the different interfaces the server component implement, this is to ask for an interface pointer through a pointer to a different interface. Because every COM interface inherits from IUnknown, every COM interface defines a QueryInterface method that allows this navigation.

There is no fixed rules for implementing QueryInterface, you can choose among if-then-else statements, hash tables, etc… as a way of selecting the appropriate pointer to be returned, this is completely up to the programmer. Nevertheless, some basic principles should be followed when implementing the function:

- The most important is that when a client asks for a pointer to IUnknown, the component has to return the same IUnknown pointer, no matter through which interface the client is asking for it. This is an essential COM principle that allows clients to check whether two different interfaces are implemented within the same components, just by querying IUnknown from two separate interfaces and comparing the returned pointers.

- A Null value has to be returned when the component does not implement an interface the client is asking for.

- The client needs to be allowed to navigate through all the interfaces regardless the starting point (e.g. the initial interface pointer the client holds)

The above paragraphs establish an interesting COM principle: *a compiled client will ask for those interfaces it knows a priori and will remain "unaware of" and "unaffected by" the rest of interfaces the server component supports*. And, in fact, it could not be in a different way, since a client will never need to know about the rest of interfaces, because once compiled there is not any way of writing code at run-time, for calling those interfaces. From the above, it follows that an existing COM component can be upgraded by adding new interfaces that implement the new required functionality. Existing applications that use the component will continue using the component through the old interfaces, while new applications can benefit from the services implemented in the new interfaces.

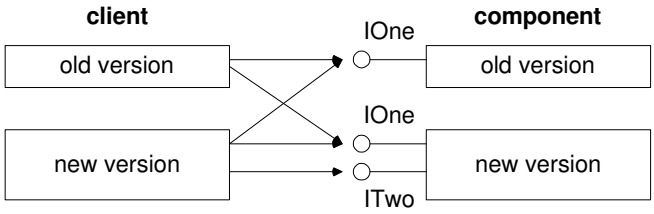


Figure 10 Usability of new COM component versions by existing and new applications

You should not even think in supplying new functionality by updating existing and already delivered and being used, COM interfaces. By definition COM interfaces are fixed once compiled, and only in extreme cases you should plan a new version of your component by modifying and recompiling the interface, but in this case you have to advise your clients about this fact, because it can potentially break their applications.

Every COM interface is characterized and identified by an IID (a universally unique identifier for the interface). This identifier is a structure that can automatically generated by compilers or specific purpose tools such as the Microsoft "Guidgen.exe". An example of IID is shown below.

```
// {7771D770-C346-11d1-8214-0080C879AC43}
static const GUID <<name>> =
{0x7771d770,0xc346,0x11d1,{0x82,0x14,0x0,0x80,0xc8,0x79,0xac,0x43}};
```

Since the interface IID is the value client will pass to QueryInterface to get the actual interface pointer back, IIDs never have to be changed. Otherwise clients will ask for pointers to interfaces no longer existing, and therefore they will not be able to access services they could access in previous versions of the component, resulting in that backwards compatibility will be lost.

If in version "n" the component needs to implement additional services to the ones included in version "n-1" an additional interface with a different IID needs to be implemented in the component.

ADDREF AND RELEASE (I.E. REFERENCE COUNTING AND THE LIFE CYCLE OF COM COMPONENTS).

The two other functions in IUnknown, i.e. AddRef and Release represent the COM strategy to control the life cycle of components.

Clients of COM components use AddRef and Release for notifying them they want to use or leave using a specific interface. Thus, a client of a COM component will never delete it, instead it will call Release on the interface it is pointing to, and the component will delete itself if its reference counting decreases to zero. This mechanism allows several clients to access the same server component at the same time without resulting in an application crash caused by the accidental of the server by one of its clients.

But still the client of COM components has some responsibilities in this memory management system.

A client should call AddRef every time it copies an interface pointer to another interface pointer, because the server component will not be aware of this action.

A client is ever responsible for calling Release on a particular interface when it no longer is going to use that interface pointer. This is true even when the component implements more than one interface (i.e. the client has to call Release in every interface pointer although they are implemented within the same component). Nevertheless, it is up to the component designer to keep a reference counting for every interface or just a global reference counting for the component.

As a general rule, the client would not need to call AddRef *when getting an interface pointer by invoking a function, because the implementation of the function would have to be done in such a way that* AddRef *is called before returning the interface pointer. This is true in* QueryInterface *and also in* CoCreateInstance *(this function will be commented later).*

The implementation of AddRef and Release could be as simple as a global reference counter that is increased in every call to AddRef and decreased in every call to Release. Besides, if a statement can control the deletion of the component when the reference counter reaches zero.

Of course there are several strategies for reducing the number of times AddRef and Release are called within a component. You can optimize the performance of your applications by omitting calling AddRef and Release in an interface pointer which lifetime is completely nested within another pointer to the same interface.

7.2.3 Creation of COM components (IClassFactory)

COM components and applications using COM components usually have to perform many routine actions, such as browsing in type libraries, loading servers or instantiating components. To ensure that these operations are performed in a standard way, a COM library exists supporting this functionality and much more (OLE32.dll).

CoCreateInstance is one of the functions exposed by the COM library and it is among the most remarkable one.

The client that needs to create a new component calls ::CoCreateInstance, passing in the class id of the component to create (CLSID) and the interface id (IID) it wants to point to.

The same way and interface is uniquely identified by an IID (interface ID), a component class is identified by a unique CLSID (class ID) and a type library is identified by a LIBID (library ID). The three one are universal unique identifiers

CoCreateInstance rather than directly instantiating the component it creates an additional component called class factory (or class object), points to one of its interfaces (very commonly IClassFactory) and invokes its method CreateInstance that actually instantiate the component.

Why use class factories instead of directly creating the component? Usually class factories are designed by the same component implementer, and they are a useful mean of encapsulating specific task that have to be done for the correct instantiation of the component. Very often class objects are implemented in the same server responsible for delivering the actual components.

Between the time lapses that begin when a client calls CoCreateInstance and ends when the component is actually instantiated, COM will connect with the right server of components and it will ask for an appropriate class factory.

CoGetClassObject is the COM function used to retrieve the appropriate class factory that knows how to instaciate the component it is being asked for. CoGetClassObject is also a function exposed by COM, so that the client can bypass CoCreateInstance, and directly invoke CoGetClassObject to get the class factory.

As it can be seen the definition of both functions is quite similar, but CoGetClassObject rather than returning a pointer to the component that needs to be created, returns a pointer to the class factory responsible for its creation. After that, the client needs to invoke ::CreateInstance through the IClassFactory pointer.

The latest creation mechanism is used when several components need to be instantiated at once (i.e. by using CoCreateInstance CoGetClassObject os called as many times as number of objects need to be instaciated while directly accessing the class factory requires only one call regardless of the number of objects). is achieved till there is another mechanism for triggering the creation of a new COM component, this is the COM function CoGetClassObject (in fact, CoCreateInstance implementation makes use of CoGetClassObject).

The second circumstance in which you have use class factories is when you want to connect to an interface different from IClassFactory (e.g. IClassFactory2), because CoCreateInstance only access class factories trough IClassFactory.

IClassFactory is a quite simple interface, with only two member functions: CreateInstance and LockServer. LockServer is a mechanism for keeping a server alive regardless of the existence of clients or not. By locking a server, this is prevented from being unloaded from memory (see ref Inside COM for a complete description of a C++ COM component implementing IClassFactory).

7.2.4 Components re-use: Containment and Aggregation

As previously mentioned, COM does not rely on implementation inheritance. Instead, specialization of COM components has to be made by containment or aggregation. Both techniques allow achieving the same objective, i.e. an outer component uses the services provided by the inner component, but using a different approach.

CONTAINMENT

By containment the outer component connects with the inner component pointing to its COM interfaces. Frequently class factories have responsibilities in initializing the group consisting of outer and inner components. Calls from a client are directly forwarded and delegated to the inner component interfaces.

The outer component can implement its own interfaces or even the same interfaces as the inner component. In the mean time, the outer component can perform some tasks before and/or after delegating the client call to the inner component/s. This technique is widely used as a mean of extending the behaviour of an interface.

No special implementation details have to be included when reusing components by containment. Since the life cycle of the inner components is completely nested within the lifetime of the outer the reference counting of the inner components can be super-seeded. The only precaution that needs to be taken into account is to design an IClassFactory able to instantiate the outer as well as the inner components.

AGGREGATION

A more specialized, although no so widely used way of re-using COM components is called aggregation. Conversely to containment, when a component aggregates other components, a client will see both, the outer and the inner components. But the interesting thing is that the client will not be able to distinguish which interfaces are really implemented by the outer and which ones by the inner component. This way, a client pointing to one of the inner component interfaces should be allowed to navigate to the interfaces implemented in the outer component.

As you can figure out, aggregation has more implications and difficulties as regards to its proper implementation than the simpler containment. First of all, it is necessary to make the client believe that both components are just one single entity. To achieve this goal, when the outer component creates the inner one, by using the traditional CoCreateInstance or IClassFactory::CreateInstance, a pointer to its IUnknown interface it is passed as an [in] argument (second argument of CoCreateInstance, or first arg. of IClassFactory::CreateInstance). This pointer is the back door for the inner component to call the member functions of the outer, because through the IUnknown pointer the inner component can query other interfaces of the outer component

As discussed earlier, a COM principle is that a client needs to be allowed to ask for an IUnknown pointer through pointers to two different interfaces and get the same response (e.g. the same IUnknown pointer).

This raises an interesting question: if from an aggregated component (i.e. the inner component) a client queries an IUnknown, he expects the returned pointer be the same as the pointer to the outer component IUnknown rather than the inner component IUnknown.

The easiest approach to solve these two circumstances is to forward every call to one of the inner IUnknown methods to the outer component IUnknown implementation. Therefore, the inner component implements what it is usually called a delegating IUnknown.

But this delegating IUnknown is only useful for components that are going to be aggregated what most of the times is now known a priori. If the component implements only a delegating IUnknown and it happens that finally the component is contained instead of aggregated, then the outer component will not be able to control the inner component because every call to its IUnknown methods will be delegated back.

Definitively this is something nobody would like to happen, and the obvious solution is to implement two IUnknown interfaces within the inner component:

- A delegating IUnknown, i.e. the implementation that a client of the component will see when the component is aggregated

- A non-delegating IUnknown, i.e. the implementation that the outer component will use to control the inner one. An external client will never see the non-delegating IUnknown.

The key point of aggregation is to let a component know when it is going to be aggregated or contained, and as previously said, this has to be done when the inner component/s is created (ref.).

7.3 OLE Automation and IDispatch.

What we have described above is strictly what a component needs to implement in order to be a full rights COM component. Automation is not something different from COM but an additional mechanism implemented on top of COM foundation.

Automation is widely used by interpreted languages, such as VB and Java, and also by applications such as Excel, Word or Access. The main advantage of automation is that it makes extremely easy to create code for calling and controlling other applications and components.

7.3.1 Dispinterfaces

An automation controller rather than in "compile-time type casting" relies on "run-time casting". While a C++ application needs the header files of the components it wants to use, so that the compiler can check types when compiling, an automation controller does not need them.

To achieve this flexibility an automation controller does not use directly the component COM interfaces, but a different interface implemented by automation servers, which name is IDispatch. To put it in few words, IDispatch is a macro that allows invoking whichever member function of the COM interfaces in a single way.

IDispatch is not a new type of interface, it is a normal COM interface that, as every COM interface inherits from IUnknown.

IDispatch is basically a mechanism for obtaining information of a component (i.e. GetTypeInfoCount, GetTypeInfo and GetIDsOfNames) plus the generic invoking mechanism (i.e. Invoke) valid for all kind of functions and a given set of valid argument types.

GetIDsOfNames and Invoke provide the key behaviour of IDispatch. An automation client uses GetIDsOfNames to obtain the DISPID of a member function (i.e. a long integer identifying a specific member function of the dispinterface), and Invoke to actually call that method.

Conversely to LIBID, IID or CLSID, a DISPID is not a uuid, every automation interface will supply its own particular DISPIDs which can be the same as those implemented by a different interface.

An automation interface (an interface derived from IDispatch), provides a common gate and a common way of invoking component methods, independently of what those methods looks like. To achieve this goal, the client does not call directly a particular method, it puts this responsibility in IDispatch::Invoke. In this sense an automation interface acts as a translator between the server and the client.

But, in this so flexible approach, the client also needs to know some information about a method before calling it, e.g. what are the arguments of that method and which are their types?

To be more precise, the client does not need to know this information in order to successfully call the method. In fact, as long as you call Invoke passing an array of arguments which types can be represented by

a VARIANTARG structure (see below), the call will be successful. Obviously when Invoke tries to call the dispinterface method it will fail and what the client will obtain is a DISP_E_XX error (an automation error).

We will not discuss here the VARIANTARG or the DISPPARAM structures in any further detail. The interested reader can find this information in most of the bibliographic references dealing with COM standards (see e.g. Inside COM, etc). Just mention that the OLE automation programmer has to be aware of the allowed types that can be represented by the VARIANT union because there is limited number of them.

7.3.2 Type Libraries

Most object-oriented programmers are used to utilize type libraries in an indirect fashion. Thus, for example, a VB programmer very often uses the Object Browser Microsoft delivers as part of the new versions of the programming environment to find out information about components he want to use or to call inside his application.

He needs this information in order to spell properly the method or property names, as well as method argument names or types. A user of OleView (included in Visual C++) is dealing with a quite similar tool.

But, the obvious question now is where all this information comes from? How a component exposes its method names, argument names and types so that such sort of tools can make use of it? The answer is "Type Libraries".

Type Libraries are binary files that can be generated by appropriate tools such as MkTypeLib or the newer MIDL compiler. These compilers generate type libraries from textual descriptions of components and interfaces, often referred as IDL files. IDL files are completely language independent. Somehow a type library is the way COM replaces header files in C++.

If you want to experiment with the many options type libraries offer, just open the VB5.0 DevStudio and add a reference in your project to TLBINF32.dll. Using the Object Browser you will see what this can offer to you.

If you want to have a more clear idea about what an IDL file looks like, launch OleView and load one component you know about. One of the cool things of OleView is that it automatically generates an IDL-like definition for you component. Even more, you can copy that IDL definition and change it slightly to generate a new type library with new interfaces (be sure you have changed GUIDS before compiling. Guidgen.exe will help you with that).

Once compiled (e.g. using MIDL.exe, also included in C++ DevStudio) the library and its contained interfaces will be ready to be used. Thus, for example, open VB5.0, create a new DLL project, open the window of references, search for the newly created type library, add a reference to it, and finally, inside your class, key Implements InterfaceName. VB5.0 will look for that type library, and will utilize it to, on-the fly, generate definitions of all member functions and properties defined in the interface.

The key idea of type libraries is to have a binary file representing all that information a client of COM components may need to make use of them. In summary, it can be said that a type library is a C++ header file that only includes the public members of the classes (i.e. interfaces). Most programming environments can interpret these binary files in the appropriate way for the language (e.g. Microsoft DevStudio will generate header files, while VB will directly supply a class skeleton).

7.3.3 Dual Interfaces

A better approach than implementing dispinterfaces or COM interfaces is to merge these two behaviours into a single interface structure, in order to get what is called dual interfaces. This approach gets the benefit of the faster access to vtbl functions (that C++ programmers will acknowledge) and the flexibility of dispinterfaces that will make automation programmers happier.

A dual interface is an interface that inherits from IDispatch rather than directly from IUnknown. By doing this, the vtbl of our custom interface will contain pointers to the 3 IUnknown methods, plus to the four IDispatch methods plus the specific function pointers of the interface. This way part of the dispinterface can be integrated within the COM part of the dual interface.

7.3.4 Performance

There is an obvious penalty associated with the flexibility of automation components and the usage of IDispatch, and this is performance. A call to a member function through the vtbl could be 100 times faster than a call to the same function through IDispatch.

7.4 Developing CAPE-OPEN Components

7.4.1 CAPE-OPEN standard releases

The CO standard for the COM platform is released using two zipped packages. The first one enclose the raw MIDL files and the second one the result of Microsoft MIDL Compiler especially the TLB library.

For example for the version 1.0.0, we have:

- CAPE-OPENv1-0-0.zip enclosing 15 IDL files (5 342 lines)
    - CAPEOPEN.idl
    - COGuids.idl
    - Common.idl
    - Parameter.idl
    - Error.idl
    - COSE.idl
    - Thrm.idl
    - Unit.idl
    - Smst.idl
    - Solvers.idl
    - Ppdb.idl
    - Reactions.idl
    - PetroleumFractions.idl
    - Minlp.idl
    - Psp.idl
- CAPE-OPENv1-0-0.tlb.zip
    - CAPE-OPENv1-0-0.tlb

- capeopen_i.c
- capeopen_p.c
- dlldata.c
- CAPE-OPENv1-0-0.h
- buildcom.bat that allows to generate the above tlb and .c/.h files from MIDL files. Produced by midl version Microsoft MIDL Compiler Version 5.01.0164 (Visual Studio 6.0). Also successful compilation with midl version Microsoft 32b/64b MIDL Compiler Version 6.00.0361 (Visual Studio .NET Beta 2003 version 7.1.2292). The command is:

```
midl capeopen.idl /nologo /server none /client none /tlb CAPE-OPENv1-0-0.tlb /h CAPE-
OPENv1-0-0.h
```

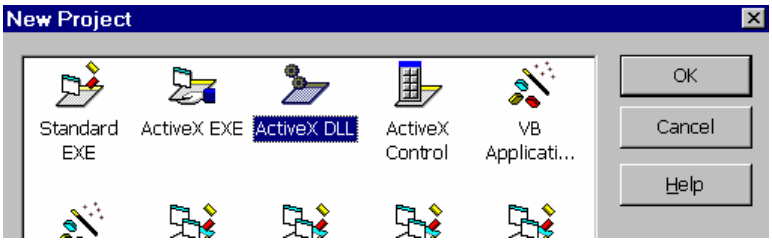
7.4.2 Basics of COM component development

In C++

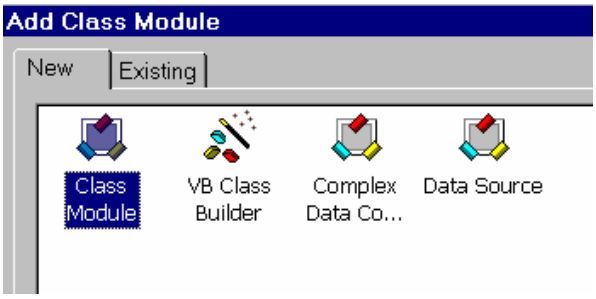
There are several ways to develop COM components in C++. For non experts, it is strongly recommended to use a helper library, such as Microsoft ATL [6].

In VB

- Create an ActiveX DLL project



- Add a class module



- Rename the project and class module. Be aware that the progID of your component will be name up of the name of the VB project plus the name of the class module. In the example below, the progID would be **Mixer.MixerCO.**

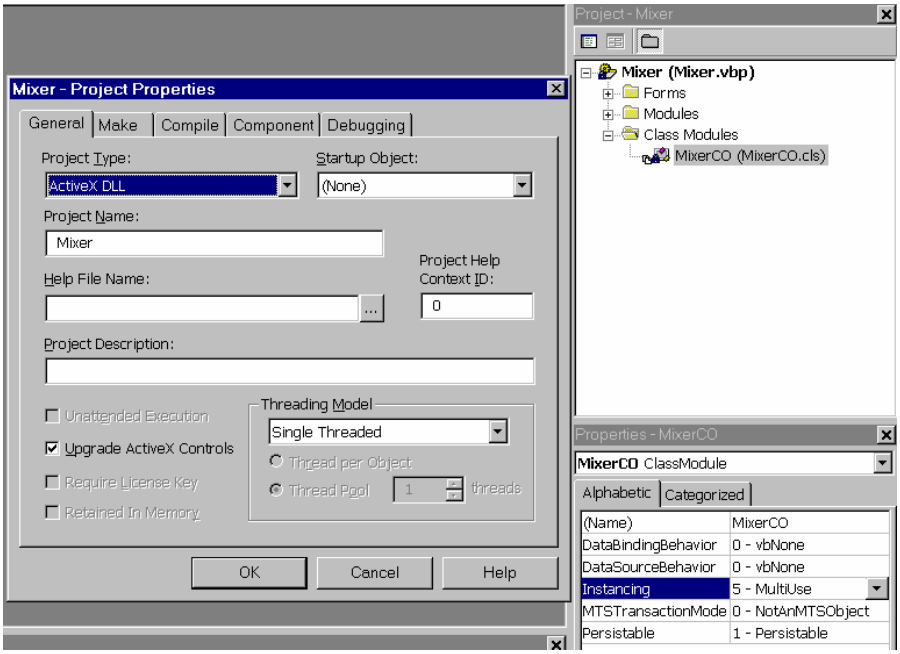


Figure 11 Setting the ProgID of a component

- Reference the CAPE-OPEN library. if the CAPE-OPEN library does not appear in the list of available references, press “Browse” button and locate the CAPE-OPENvX-X-Xtlb

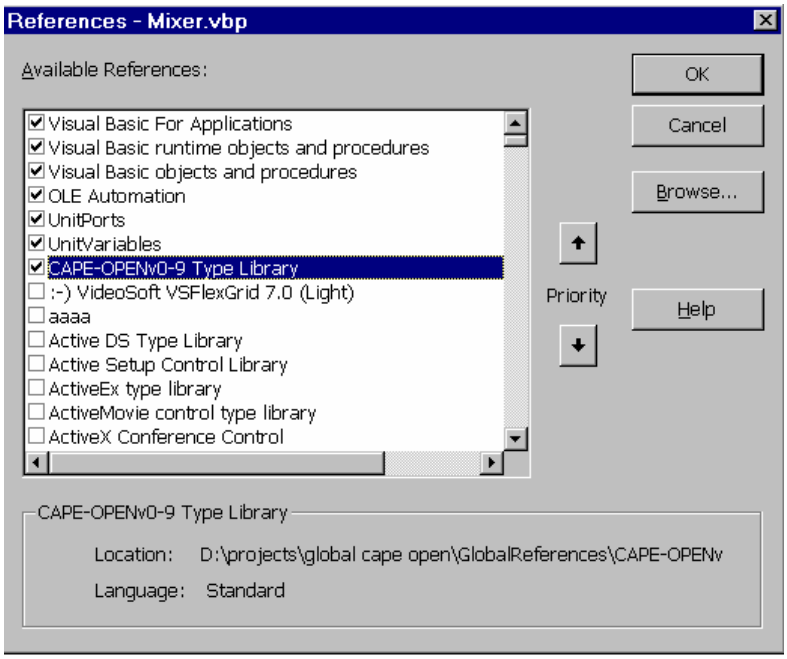


Figure 12 Referencing the CAPE-OPEN library

- Use the Implements keyword for each desired CAPE-OPEN interface.

```
Option Explicit

'Cape Open Unit Interface implementation
Implements ICapeUnit
Implements ICapeUnitEdit
Implements ICapeIdentification
```

- Implement each method of each interface.

```
Private Property Get ICapeIdentification_ComponentName() As String

End Property
```

```
Private Function ICapeUnit_Calculate(message As String) As Boolean

End Function
```

```
Private Function ICapeUnitEdit_Edit(message As String) As Boolean

End Function
```

Where to place the CAPE-OPEN type library and how to register it

Technically, it doesn't matter where to place the file, but GCO will recommend

\Program Files\Common Files\CAPE-OPEN

You don't need to register it if you have installed a CAPE-OPEN compliant simulator. If you haven't you can register it with *REGTLIB.EXE CAPE-OPENv0-9.tlb*

If it's not registered, it won't appear in list of Figure 12.

HOW TO MAKE A SIMULATOR AWARE OF NEW CAPE-OPEN COMPONENTS

How to register a CAPE-OPEN component

It's not enough registering it with regsvr32 (or the automatical registration that VB performs after compiling), because the CAPE-OPEN categories must also be assigned to the components.

One way is double clicking on a CLSID.reg file as showed in Figure 13.

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{434DA84A-0EF6-11D4-A3DE-00902724BD35}]
@="HYP/Mixer-Splitter"[4]
```

[4] Not required by the CAPE-OPEN standard, but useful for some COM browser applications.

```
[HKEY_CLASSES_ROOT\CLSID\{434DA84A-0EF6-11D4-A3DE-00902724BD35}\CapeDescription]
@=""
"Name"="HYP/Mixer-Splitter"
"Description"="Mixer Splitter with multiple inputs and outputs"
"CapeVersion"="0.9"
"ComponentVersion"="1.0.2"
"VendorURL"="http://www.hyprotech.com"
"HelpURL"="file://<ProgramFilesDir>\\AEA Technology\\CAPE-OPEN\\CO Mixer-
Splitter\\RELEASE NOTES.doc"
"About"="Hyprotech European HQ\\nPg. de Gràcia, 56\\n08007 Barcelona\\nSpain\\nPhone:34-
93-215-6884\\nFax: 34-93-215-4256"

[HKEY_CLASSES_ROOT\CLSID\{434DA84A-0EF6-11D4-A3DE-00902724BD35}\Implemented Categories]

[HKEY_CLASSES_ROOT\CLSID\{434DA84A-0EF6-11D4-A3DE-00902724BD35}\Implemented
Categories\{678C09A1-7D66-11D2-A67D-00105A42887F}]
[HKEY_CLASSES_ROOT\CLSID\{434DA84A-0EF6-11D4-A3DE-00902724BD35}\Implemented
Categories\{678C09A5-7D66-11D2-A67D-00105A42887F}]
```

Figure 13 Sample COM Registration entries

There are more sophisticated ways to register your CAPE-OPEN components that double clicking on a .reg file. All of them will finally consist in registering the above registry entries. However, since the aim of this part is to describe the CAPE-OPEN requirements, rather than the COM mechanisms, this part will focus on working with this .reg file.

Let's review the file. In Figure 13, {434DA84A-0EF6-11D4-A3DE-00902724BD35} is the CLSID of the registered component. Next sections explain how to know the CLSID of your own component.

The file adds registry sub-entries (information) to the registration of your component in the windows registry.

The 'CapeDescription' entry adds user-friendly description to the component.

The 'Implemented Categories' assigns categories to your component, according to the following table:

```
CAPE-OPEN Component                    {678c09a1-7d66-11D2-a67d-00105a42887f}
CAPE-OPEN Thermo Routine               {678c09a2-7d66-11D2-a67d-00105a42887f}
CAPE-OPEN Thermo Property System       {678c09a3-7d66-11d2-a67d-00105a42887f}
CAPE-OPEN Thermo Property Package      {678c09a4-7d66-11D2-a67d-00105a42887f}
CAPE-OPEN Unit Operation               {678c09a5-7d66-11D2-a67d-00105a42887f}
CAPE-OPEN Thermo Equilibrium Server    {678c09a6-7d66-11D2-a67d-00105a42887f}
```

Figure 14 GUIDs for CAPE-OPEN Component categories

How do I know the ProgID of my component?

- In C++

The developer must enter manually the ProgID. So, you must check your source code.

- In VB

The progID is name up of the name of the VB project plus the name of the class module. See Figure 11

How do I know the clsid of my component?

- In C++

The developer must enter manually the CLSID. So, you must check your source code.

- In VB

VB assigns automatically a CLSID every time you compile a COM component. You can only look it up after the component is compiled. There are Two alternatives:

- Oleview.exe (it's a Visual Studio tool)

go to ObjectClasses\AllObjects

Look for the Progid of your component , Mixer.MixerCO (if .reg file has not yet been executed) or "`HYP/Mixer-Splitter`" (after running succesfully your .reg file).

The name "`HYP/Mixer-Splitter`" is registered by one of the first lines of the .reg file.

To get the CLSID, right click on your component and choose "copy CLSID to clipboard"

- Browse through the windows registry (more complex)

Run regedit.exe

go to branch HKEY_CLASSES_ROOT\CLSID

Select menu Edit\Find\

Type the progid of your component.

The clsid is the name of the parent of the progID branch

VB6 SHORTCOMINGS

Since VB changes the CLSID of your component everytime you compile, the developer should replace the CLSID value in Figure 13 every time you compile.

To avoid that, you'll find how to set the compatibility option to keep the CLSID fixed everytime the component is compiled, which will prevent you from having to edit this .reg file everytime.

How to keep fixed the CLSID of a COM component

(i) Compile once your component.

(ii) from outside of VB, take a copy of your target dll file (eg target.dll) to the same path (eg. Name it target_ref.dll). You have created a reference dll.

(iii) From VB, select the project that makes up the CAPE-OPEN component (in case you're browsing a set of projects)

(iv) Select menu "project"\"<your project> properties"

(v) Go to tab "Component".

(vi) Set "Version compatibility" option to "binary compatibility".

(vii) Type in the name of the reference dll (eg. target_ref.dll)

(viii) Compile (make) again to test everything went ok.

What's a "reference dll"?

You have created a reference dll of your COM component which will "never" be changed.

Setting the compatibility option, every time you compile and make a new dll, VB will assign the same CLSID to your component (getting the CLSID) from your original reference dll. Keep in mind that VB can only keep the same CLSID if the interface of the component has not changed.

It is advisable to be aware of all the concepts required to develop COM components in VB. A part from the VB online documentation, there are a lot of books in the market about this topic (important concepts are: COM component, progid, clsid and dll).

Edit your .reg file only once

After following steps from "How to keep fixed the CLSID of a COM component", you must edit the .reg file

Edit the clsid.reg replacing EVERYWHERE {434DA84A-0EF6-11D4-A3DE-00902724BD35} with the CLSID of your component. FOR YOUR CAPE-OPEN SIMULATOR TO BE ABLE TO FIND THE COMPONENT, YOU WILL ALWAYS HAVE TO DOUBLE CLICK ON THE CLSID.REG AFTER YOU START or STOP DEBUGGING IT FROM VB.

Since the CLSID will not change any more, why isn't it then sufficient to double-click the clsid.reg ONE time?

The reason of this shortcoming is that VB6 removes the category information from your component AFTER YOU CLICK START or you STOP. And without the category information, a CAPE-OPEN simulator will not find your CAPE-OPEN component in the windows registry.

The reason is that, when you debug, VB6 maps temporarily your component progId to its own dll to be able to debug it properly. When you press stop VB6 restores automatically the mapping from your component progId to you memphis.dll. In this process of un/registering, VB6 looses the registered categories for your component.

DEVELOPMENT TROUBLESHOOTING

My Simulator cannot find my registered CAPE-OPEN component anymore

Be aware that, in a machine, you can only have installed once a component with a given ProgId or CLSID. For instance, if you install the binary MixerSplitter component distributed by Hyprotech and you also compile and make the source code of this component, only one of these two components will remain registered (the one that was registered last).

To avoid this, you should better change the CLSID, ProgID and the name of the compiled component. How to change each of them:

- **CLSID**: remove the "binary compatible" setting and repeat steps in 0. You must also redo step 0.
- **ProgID**: In VB, as explained in 0, you should change the name of the project or the name of the class module.
- **User friendly Name**: Change the following lines of .reg file.

```
@="HYP/Mixer-Splitter"

"Name"="HYP/Mixer-Splitter"
```

7.5 Component deployment

7.5.1 What to do to "deploy/distribute" a new CAPE-OPEN unit?

You can use an install application such as Installshield. See 7.5.2 for a practical guide on the use of Installshield Express

You can also use the "package & deployment wizard", which is distributed with MS Visual Studio (it's in "Windows start button"\"MS VisualStudio\tools).

7.5.2 Usage of InstallShield for CO components

Guidelines to prepare a setup for a CAPE-OPEN COM component with InstallShield Express 2.0, by InstallShield Corporation. Still, some mentioned hints are useful for any kind of setup technology.

These guidelines are structured following the Checklist GUI of the software. Each one of the following sections refers to one of the dialogs of the application that are used to configure each one of the checklist actions.

Installshield Express allows to:

- Copy as many files as your CAPE-OPEN component requires (including release notes, ...)
- Register the COM DLLs, and adding any kind of information to the Windows registry
- Prepare a consistent uninstall procedure.

SET VISUAL DESIGN

App Info

"Default Destination Dir(DDD)" is made up of concatenating the previous fields (resulting in ProgramFilesDir\Company\AppName).

If you want to break this rule, set the DDD AFTER setting the company or the Application Name.

The Application Name cannot contain a backslash (\). Otherwise, in the registry key under (`[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\`),

It is interpreted as a subfolder and "Control Panel/Add-Remove Programs" won't display the entry. So, it would then be impossible to uninstall the component.

The DDD cannot contain a /. So, if the Application Name contains a /, take care it's not transmitted automatically to the DDD

Features

Select the 'automatic uninstaller' checkbox.

SPECIFY INSTALLSHIELD OBJECTS

InstallShield can automatically determine which Installshield Objects your application requires. We don't use this option, but we specify the required files manually.

SPECIFY COMPONENTS AND FILES

Groups

Create a Group for each set of files that share the same destination directory.

We have used:

Program Files

Normal files required by the component. Install in <INSTALLDIR>.

Type Libraries

COM type libraries, such as the CAPE-OPEN0-9.tlb. Install in <INSTALLDIR>.

System Files

System files required by your component. They will be copied on the Windows system directory and will upgrade the whole Windows System. Install in <WINSYSDIR>

Common Files

Imagine you plan to distribute several CAPE-OPEN components (CO1 & CO2)that share a common COM component(CCC). It would not be wise to install the CCC component in <INSTALLDIR>, since a COM component can only be registered in a single windows directory at a time. That means that, if you unregister or remove the files from CO1, you might unregister CCC from the system, and then CO2 would not work.

So, it is recommended to place them in <CommonFilesDir> (C:\Program Files\Common Files\Vendor Shared in my case)

Insert the required files for each group. Click on properties and select 'Allow Express to Self-Register this file' for the files that contain COM components that require to be registered. The DLL of your CAPE-OPEN components and some System files require this action.

MAKE SYSTEM FILE CHANGES

Not required for our examples (fortunately!)

SELECT USER INTERFACE COMPONENTS

Dialog Boxes

select program folder

If you want to place some links in the pop up menu accessible through "Windows Start button"\ "Programs", enter the program folder name. We use the standard CompanyName\CAPE-OPEN\ComponentName.

Although when building the disks there's a warning about the above path being too long, something as long as "AEA Technology\CAPE-OPEN\HYP CO Kit v1.0" work properly in Win98 & WinNT.

MAKE REGISTRY CHANGES

CAPE-OPEN components need the following registry entries:

- Categories: For the COSES to find them easily in the registry.
- Static description of the software components: Description, version, about,...

If your components don't add these entries within DllRegisterServer procedure of you component, you can do with the help of your setup application.

Registry-keys/Registry value

Not recommended, since it has some bugs and it's difficult to maintain.

REG files

The easiest way to register categories and the CapeDescription folder is adding the corresponding .REG file to this tab . See Figure 13 for a sample of the required registry entries.

SPECIFY FOLDERS & ICONS

Add here the links that you want to appear in the Program Folder ("Windows Start button"\ "Programs"), such as release notes, test cases, ...

How to add an uninstall icon to the program folders

If you want to add uninstall icon, add the icon with these parameters

Run command: <WINDIR>\uninst

Run command parameters: -f"<INSTALLDIR>\DeIsL1.isu"

Description: Name of the icon

RUN DISK BUILDER

If there's an error complaining that a file cannot be found, it could be that the file is open by another program, such as Word, that locks the file.

COPY TO FLOPPY

If you choose the option 'Path for a 1 File installation', it will generate a single self-extractable file (useful to send by e-mail or publishing in a Web page).

8. CORBA-specific architectural and technical issues

On the COM side, the definition and the distribution of a common file for the CO COM specifications have been decided. This file is a compiled version of the COM IDL sources, required by the MS-Windows operating systems. It is named *CAPE-OPENvX-X-X.tlb*. This COM TLB is versioned using the build-in COM versioning system. There is a single COM TLB for all the CO COM interfaces. An equivalent functionality has to be found for the CO CORBA specification.

We present below some technical alternatives and final selected options corresponding to each issue. Some technical information are provided. As a constraint, we want to be close to the solutions (when they exist and when they are acceptable) coming from the COM side.

8.1 Format release

The COM side makes public not only the IDL files but the type library which is the compiled version of these files. This choice is feasible since COM is a proprietary technology.

From a CORBA view, we could supply a corresponding library but this library will depend on the ORB software supplier, the implementation language and the operating system. We would be obliged to supply several libraries.

Somehow that is not the philosophy of the CORBA technology, the standard is at the level of IDL since the language mappings are standardised. Additionally, we would limit the CO standard to certain ORB implementation which is a strong contradiction to the aim of implementation independency.

The CO standard for CORBA platform is released using raw IDL files.

8.2 File system

The alternatives are:

- x file: one file per topic
- 3 files: one file for each types of interface: common/business/COSE
- 1 file: one file for all.

In order to be closer to the COM type library (1 file), we choose the last alternative. Furthermore, we decide to have a high-level scope (e.g. CORBA module) named CapeOpen for all CO CORBA specifications. So due to the scoping rules in CORBA one file is mandatory.

Even so we should note that this single file involves some drawbacks at the implementation level such as for instance to get an arduous generated code, to have some useless generated code and to get big size generated files. But this drawback is not a real problem in praxis. Normally, the application developer does not have to look into the generated stub/skeleton code and most intelligent linkers will eliminate such kind of dead code. On the other hand, no include statements are necessary thereby simplifying the handcrafted code which is much more interesting. Furthermore one IDL file simplifies the handling of the interface definitions in several ways:

- There is only one include directive necessary,
- The maintenance of the IDL itself is simpler as we do not have to keep track of a lot of files,

- If the developer has to manage only one IDL file he can be sure that is versioning consistent. This is not the case for several IDL files. This could cause some problems during prototype development,
- Easy dissemination,
- and easy version management.

8.3 Versioning system

Contrary to the COM technology where the COM type library is versioned which allows to be updated without invalidating all software using a previous version, the CORBA technology has no built-in versioning capacity even if CORBA 3.0 gives some solutions.

Consequently, the future IDL development should make possible the downwards compatibility manually at the design level. Without giving a lot of details, some ways are addition of methods, interfaces, use of inheritance, … That will be investigated when necessary.

The OMG directive #pragma version allows giving a version number to the repository id. It isn't used here and somehow it doesn't solve the problem.

We could imagine a version number per specifications topics. But to be basic and to be close to the COM choice, one version number characterises the whole CORBA IDL file. Comment lines at the beginning of the IDL file display the version number of the CO CORBA specification such as:

```
/* IMPLEMENTATION SPECIFICATION VERSION
Type = CAPE-OPEN IDL library for CORBA platform
Version number = 1.0.0
Delivering date = March 2003
*/
```

Additionally, following this number the version number corresponding to the compliant CORBA specification is displayed. At present the final IDL file is compliant with the CORBA specification 2.0 (The Common Object Request Broker: Architecture and Specification 97-02-25; see the omg web site) and upper. The versioning should also reflect clearly the downwards compatibility of the IDL definitions. At least a comment in the file should point out compatibility such as:

```
/* COMPATIBILITY VERSION
CORBA Specification version number with which this file is compliant = 2.0 and upper
Visit the web site of the CORBA standard at www.omg.org
*/
```

It is worth noting that we can identify some limits of our previous decisions . For example, let us suppose the Smst topic wants to use the *Value Type* introduced since the CORBA 2.3.1. Then not only the Smst part but also the whole CO CORBA IDL has to be compliant with the CORBA 2.3 instead of 2.0. This jump has significant consequences on the implementation and deployment phases. We simplify the standard life cycle management but we loose some kind of flexibility. However solutions exist and M&T group will work on that if necessary.

8.4 Scoping strategy

It is decided to have a high level scope (module CapeOpenXXX). Indeed CORBA allows having nested modules which give an additional encapsulating level, in opposition to COM where all the specifications have a same and global scope. For information, the CORBA module is mapped to the C++ namespace and to the Java package according to the Language Mapping Specifications. Hence we can note two alternatives:

- One module CapeOpenXXX with no nested modules
- One module CapeOpenXXX with nested modules. It remains to intend the number of module levels

This choice is not so obvious, as a matter a fact that has important effects on the CO CORBA component development. To keep the second alternative brings to a better design but the implementation is more complex because the developer will be obliged to manage scoping due to the CO standard. Their use is recommended for a good development (almost mandatory to Java developers).

More over some C++ ORB software map the module to the class instead of the namespace (for example Inprise VisiBroker 3.x) since few compilers currently support the namespace concept. Then the developer looses the namespace functionality such as the using directive.

In spite of these drawbacks, in order to have an additional encapsulation level we recommend the use of nested modules. Even if we will be far from the COM type library, we want to use this encapsulating ability knowing that some solutions exist for the bridging COM-CORBA.

The final IDL file has nested modules organising Common, Business and COSE modules within the high level CAPE-OPEN scope.

8.5 Comment lines

Comments are inserted to give some generic information (for instance versioning, module, interface). No method is documented in order to avoid repetitive information (and so possible inconsistencies) with the Interfaces Descriptions part enclosing in the Open Interface Specification documents.

8.6 CORBA IDL file overview

As illustration, here is below the skeleton of the final CORBA IDL file version 1-0-0. This file is called CAPE-OPENv1-0-0.idl (August 2003).

```
/* IMPORTANT NOTICE
(c) The CAPE-OPEN Laboratory Network, 2002.
All rights are reserved unless specifically stated otherwise

Visit the web site at www.colan.org

This file has been edited using the editor from Microsoft Visual Studio 6.0
This file can viewed properly with any basic editors and browsers (validation done under
MS Windows and Unix)
*/

/* IMPLEMENTATION SPECIFICATION VERSION
Type = CAPE-OPEN IDL library for CORBA platform
Version number = 1.0.0
Delivering date = August 2003
*/

/* COMPATIBILITY VERSION
CORBA Specification version number with which this file is compliant = 2.0 and upper
Visit the web site of the CORBA standard at www.omg.org
*/

// This file was developed/modified by JEAN-PIERRE BELAUD for CO-LaN organisation -
August 2003
```

```
// ============================================================================
// IMPLEMENTATION SPECIFICATION FOR CORBA PLATFORM
// ============================================================================


// ---- The global scope is defined by a CAPEOPEN100 Module -------------------
module CAPEOPEN100 {



      // ---- The scope of the common interfaces --------------------------------
      module Common{


            // ---- The scope of the types and undefined values -------------------
            // Reference document: Types and undefined values
            module Types{

                  // elementary type definitions
                  typedef long        CapeLong;
                  typedef short       CapeShort;

                  typedef double            CapeDouble;
                  typedef float       CapeFloat;

                  typedef boolean           CapeBoolean;

                  typedef char        CapeChar;
                  typedef string            CapeString;
                  typedef string            CapeDate;
                  typedef string            CapeURL;

                  typedef any               CapeVariant;
                  typedef Object            CapeInterface;

                  // sequence definitions
                  typedef sequence<CapeLong>      CapeArrayLong;
                  typedef sequence<CapeShort>     CapeArrayShort;
                  typedef sequence<CapeDouble>    CapeArrayDouble;
                  typedef sequence<CapeFloat>     CapeArrayFloat;
                  typedef sequence<CapeChar>      CapeArrayChar;
                  typedef sequence<CapeString>    CapeArrayString;
                  typedef sequence<CapeBoolean>   CapeArrayBoolean;
                  typedef sequence<CapeDate>      CapeArrayDate;
                  typedef sequence<CapeURL>        CapeArrayURL;
                  typedef sequence<CapeVariant>   CapeArrayVariant;
                  typedef sequence<CapeInterface> CapeArrayInterface;

                  // Definition of CapeValidationStatus type
                  typedef enum eCapeValidationStatus{
                        CAPE_NOT_VALIDATED,
                        CAPE_INVALID,
                        CAPE_VALID
                  } CapeValidationStatus;

                  typedef sequence<CapeValidationStatus> CapeArrayValidationStatus;

                  // Definition of Undefined values
                  const  CapeLong     CapeLongUNDEFINED   =-2^31;
                  const  CapeShort    CapeShortUNDEFINED  =-2^15;

                  const  CapeDouble   CapeDoubleUNDEFINED =-4.9E-324;
                  const  CapeFloat    CapeFloatUNDEFINED  =-1.4E-45;

                  const  CapeChar     CapeCharUNDEFINED   ='\0';
                  const  CapeString   CapeStringUNDEFINED ="";
                  const  CapeDate     CapeDateUNDEFINED   ="";
                  const  CapeURL            CapeURLUNDEFINED    ="";

                  #define CapeArrayLongUNDEFINED         NULL;
```

```
            #define CapeArrayShortUNDEFINED         NULL;

            #define CapeArrayDoubleUNDEFINED  NULL;
            #define CapeArrayFloatUNDEFINED         NULL;

            #define CapeArrayCharUNDEFINED          NULL;
            #define CapeArrayStringUNDEFINED  NULL;
            #define CapeArrayDateUNDEFINED          NULL;
            #define CapeArrayURLUNDEFINED           NULL;


        }; // END Types module ---------------------------------------------


        // ---- The scope of the error interface -----------------------------
        // Reference document: Error Common Interface
        module Error{

…
        }; // END Error module ---------------------------------------------


        // ---- The scope of the identification interface --------------------
        // Reference document: Identification Common Interface
        module Identification{

…

        }; // END Identification module ------------------------------------


        // ---- The scope of the collection interface ------------------------
        // Reference document: Collection Common Interface
        module Collection{

…

        }; // END Collection module ----------------------------------------


        // ---- The scope of the utilities interface -------------------------
        // Reference document: Utilities Common Interface
        module Utilities{

…

        }; // END Utilities module -----------------------------------------


        // ---- The scope of the parameter interface -------------------------
        // Reference document: Parameter Common Interface
        module Parameter{

…

        }; // END Parameter module -----------------------------------------


        // ---- The scope of the persistence interface -----------------------
        // Reference document: Persistence Common Interface

…

        }; // END Persistence module ---------------------------------------

    }; // END Common module -------------------------------------------------


    // ---- The scope of the COSE interfaces ---------------------------------
```

```
module Cose{

       // ---- The scope of simulation context interface ---------------------
       // Reference document: Simulation context interface
       module SContext{

…

       }; // END SContext module ----------------------------------------------

}; // END Cose module ------------------------------------------------------


// ---- The scope of the Business interfaces -----------------------------

module Business{


// ---- The scope of the Physical Properties interfaces -------------------

       module PhyProp{

              // ---- The scope of thermodynamic and physical properties interface
              // Reference document: Thermodynamic and physical properties
              // and Petroleum Fractions
              module Thrm{

                     // ---- Cose module ----------------------------------------
–
                     module Cose{

…

                     }; // END Cose module ------------------------------------
–

                     // ---- ThermoSystem module ---------------------------------
–
                     module ThermoSystem{

…

                     }; // END ThermoSystem module -----------------------------
–

                     // ---- CalculationRoutine module ---------------------------
–
                     module CalculationRoutine {

…

                     }; // END CalculationRoutine module -----------------------
–

                     // ---- EquilibriumServer module ----------------------------
–
                     module EquilibriumServer {

…

                     }; // END EquilibriumServer module ------------------------
–


              }; // END Thrm module ----------------------------------------
```

```
                // ---- The scope of the Reactions interfaces ------------------
                // Reference document: Chemical Reactions
            module Reactions{

…

                }; // END Reactions module

                // --- The scope of the Physical Properties Data Base interfaces
                // Reference document: Physical Properties Data Base
                module Ppdb{

…

                }; // END Ppdb module


             }; // END PhyProp module ---------------------------------------------


             // ---- The scope of the Numeric interfaces --------------------------

             module Numeric{


                // ---- The scope of numerical solvers interface ------------------
                // Reference document: Numerical Solvers and Partial Differential
Algebraic Solvers
                module Solvers {

                       // ---- The scope of the Eso ---------------------------------
-
                       module Eso{

…

                       };  // END Eso module ---------------------------------------
-

                       // ---- The scope of the PDA ESO -----------------------------
-
                       module PdaEso{

…

                       }; // END PdaEso module ------------------------------------
-

                       // ---- The scope of the Model -------------------------------
-
                       module Model{

…

                       }; // END Model module -------------------------------------
-


                       // ---- The scope of the Solver ------------------------------
-
                       module Solver{

…

                       }; // END Solver Module ------------------------------------
-


                }; // END Solvers module ----------------------------------------
```

```
                    // ---- The scope of the Optimisation interfaces ------------------
                    // Reference document: Optimisation
                    module Minlp{

…

                    }; // END Minlp module ------------------------------------------


                    // ---- The scope of PEDR interfaces ----------------------------
                    // Reference document: Parameter Estimation and Data Reconciliation
                    module Pedr{

…

                    }; // END Pedr module -------------------------------------------

             }; // END Numeric module ---------------------------------------------


             // ---- The scope of the Unit Operations interfaces -------------------

             module UnitOp{


                    // ---- The scope of unit operation interface ---------------------
                    // Reference document: Unit Operation
                    module Unit{

…

                    }; // END Unit module -------------------------------------------


             }; // END UnitOp module -----------------------------------------------


             // ---- The scope of the Other interfaces -----------------------------

             module Other{


                    // ---- The scope of smst interface -------------------------------
                    // Reference document: Sequential Modular Specific Tools
                    module Smst{

…


                    }; // END Smst module -------------------------------------------


                    // ---- The scope of PSP interface --------------------------------
                    // Reference document: Planning and Scheduling Interface
Specification
                    module Psp{

…

                    }; //End Psp module ------------------------------------------


             }; // END Other module ------------------------------------------------


      }; // END Business module ------------------------------------------------

}; // END CAPEOPEN100 module ---------------------------------------------------
```

9. Common Interfaces

This part gives an overview of the CAPE-OPEN *Common Interface*s. It sets out a proposal for a cross concept: the CO *Common Interface*s.

The *Common Interfaces* are interfaces and implementation models for handling concepts that may be required by any *COSE Interfaces* and *Business Interfaces*. Any *Common Interface*, *COSE Interface* and *Business Interface* are described through an open interface specification document.

One of the objectives of Methods & Tools group is to provide reusable interfaces for the CAPE-OPEN interface designers to be able to concentrate on engineering concepts and not on plumbing details. There is a set of simple unrelated functionalities that would be useful for any kind of Process Modelling Component, since it would allow maximum integration between Process Modelling Components and any Process Modelling Environment to which provide services.

These recommendations are especially dedicated to the designers of CO interfaces. They introduce the *Common Interface*s and explain how to integrate them within the *Business Interface* and *COSE Interface* that the designers develop. However the developers of CO components will find useful information for understanding the *Common Interfaces* that they will have to implement.

These recommendations and the related concepts come from a conceptual approach and are independent to the distributed platform such as (D)COM and CORBA

Therefore this section 9 on how to integrate the *Common Interfaces* differentiates the interactions with *PMC objects* from *Business Interfaces* and with *PME objects* from *COSE Interfaces.* Indeed the need of integrating the *Common Interfaces* is different, the *PME objects* requires only providing the error handling strategy. While in regards to the services coming from the *Common Interfaces*, two kinds of *PMC objects* are defined, **primary and secondary**. That allows identifying the role and the scope of a *PMC object*. Thus according to its kind, the *Common Interfaces* services that the *PMC object* provides are specified.

The part defines which common services have to be proposed by the *PMC objects* and *PME objects*.

9.1 Common interfaces and COSE interfaces

This section defines the *Common Interfaces* that any PME object has to provide to any PMC. In fact this requirement is straightforward, the PME objects have to integrate the CO error handling strategy.

9.1.1 Use-case diagram

All *PME objects* have to respect the error handling strategy as shown in the below diagram.

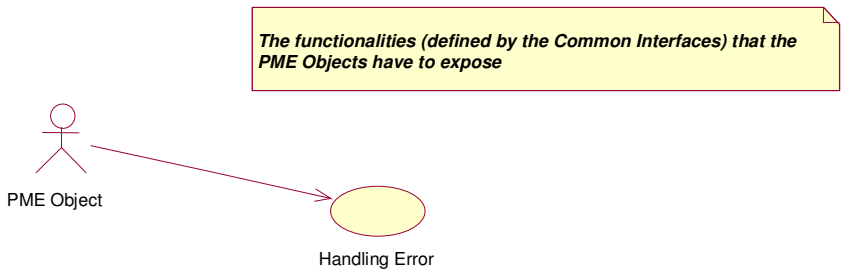


Figure 15 Use-case diagram

All *PME objects* depend on the specification document: Error Common Interface.

9.1.2 Component diagram

The following diagrams show *PME objects* (rectangles) and the interfaces (circle-ended lines) which may be provided by each object. The interfaces written in bold are described in the *COSE Interfaces* specification documents while the others in the *Common Interfaces* specification documents.

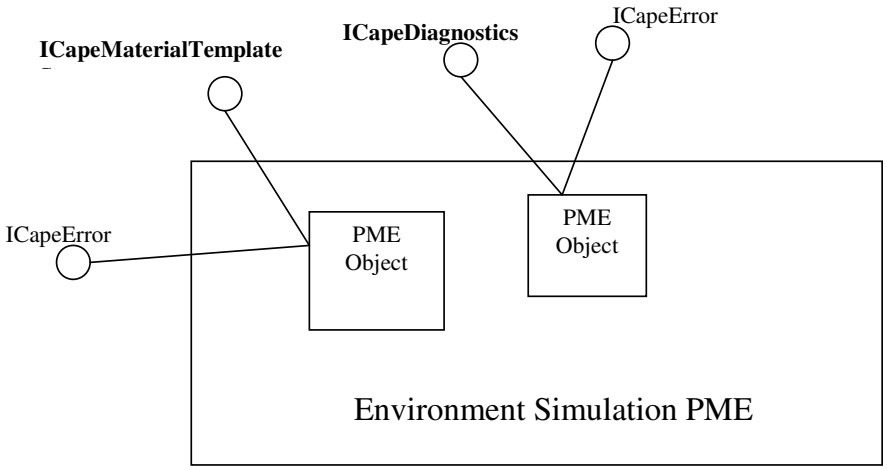


Figure 16 Simulation Environment Component Diagram

9.2 Common interfaces and business interfaces

We distinguish two kinds of *PMC objects*: *PMC primary object* and *PMC secondary object.* Thus the services they offer to any *PME* are clearly identified.

9.2.1 Primary and secondary (interface) object

PMC primary object: When a *PME* requires some kind of external functionality, it checks with the help of the CAPE-OPEN categories which suitable components are available in the system. The user will then select one of those and the *PME* will create a *CO object* that will expose the required CO interfaces. We will call *PMC primary object* to this instance which is created explicitly by the *PME*. The only exception to this rule are those objects created through a System (Factory like) interface. That is the case of ICapeThermoSystem, which may create property package instances on behalf of a *PME*. This means that each *Business Interfaces* Specification document will have to specify which are its primary objects.

PMC secondary object: Adding other *objects* is the normal way to design a *PMC* from an object-oriented approach. All the services provided by the *PMC* can not be (in many cases) present within one single *PMC primary object*. Therefore a *Business Interface* has *PMC secondary objects* which do not need to make available the same *Common Interfaces* services. For instance, a Unit Operation to expose its ports provides a way for *PMEs* to have access to a set of port object instances. These objects are *PMC secondary objects* and are only related to base *Common Interfaces* such as error handling and identification.

An advantage of interface models (such as the component paradigm) consists in the fact that interface clients get abstracted from the actual objects that implement the functionality. However, it is important to classify these two kinds of objects because *PMEs* will give them different treatments. For instance, it makes sense that any *PMC primary object* satisfies Edit and Persistence interfaces, since it is comfortable for *PMEs* to have a single point of entry for configuring and persisting the state of a *PMC*.

9.2.2 Use-case diagram

All *PMC objects* (primary and secondary) have to provide *Common Interfaces* services to any *PME* such as an identification process and an error handling strategy.

A *PMC Primary object* has to expose specific *Common Interfaces* services such as a persistence management. The following diagram summarises the mandatory functionalities that *PMC objects* have to supply.

There is no mandatory recommendation for the *PMC Secondary Object* but it could also provide services coming from the *Common Interface*s. This design choice is up to the designers of CO interfaces. For instance a *PMC Secondary Object* could provide Collecting Items functionality.

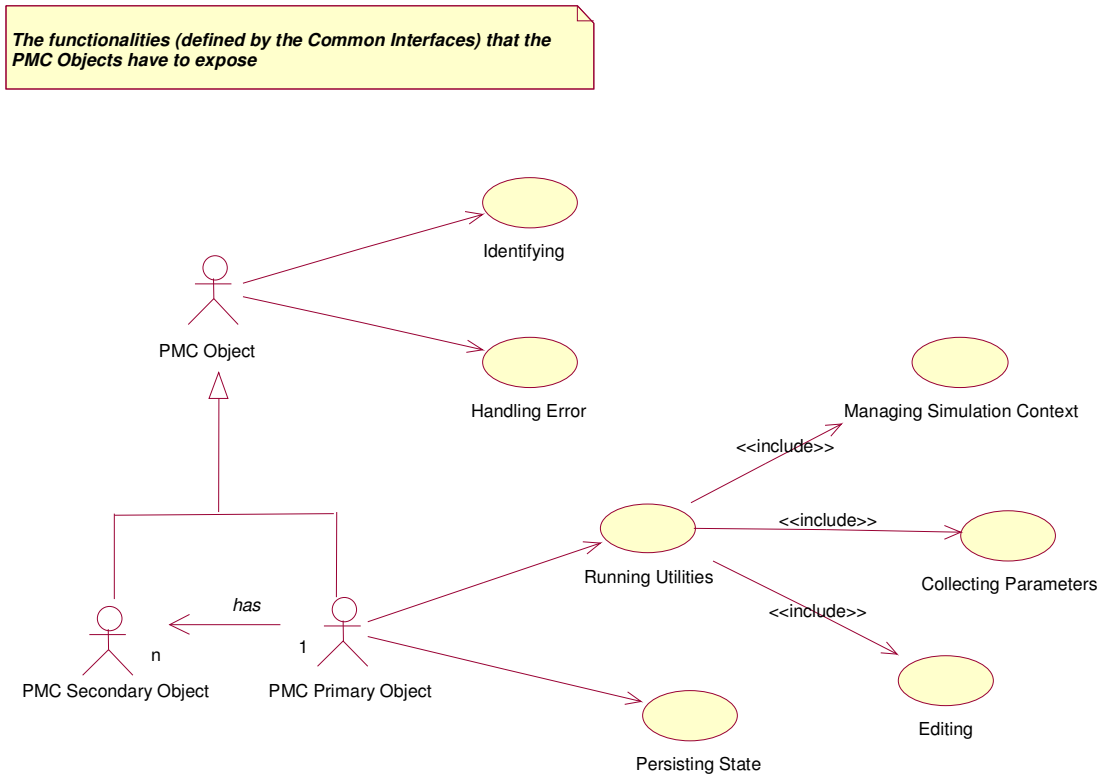


Figure 17 Use-Case diagram

Based on the previous diagram, we can identify the *Common Interfaces* documents according to its scope:

All *PMC Objects* depends on the following specification documents:

- Error Common Interface
- Identification Common Interface

All *PMC Primary objects* depends on the following specification documents:

- Persistence Common Interface
- Utilities Common Interface

9.2.3 Component diagrams

As illustration, the following component diagrams show the primary and secondary concepts for some *PMCs*.

The diagrams show the *PMC objects* (rectangles) and the interfaces (circle-ended lines) which may be provided by each object. The interfaces written in bold are described in the *Business Interfaces* specification documents while the others in the *Common Interfaces* specification documents. Note only few secondary objects are drawn.

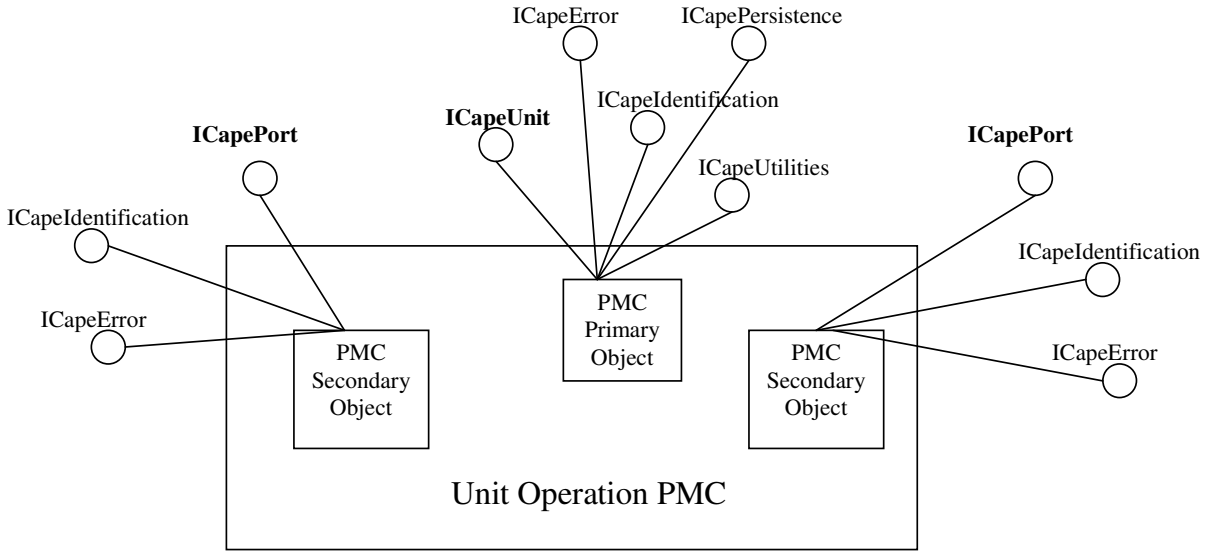


Figure 18 Unit Operation Component Diagram

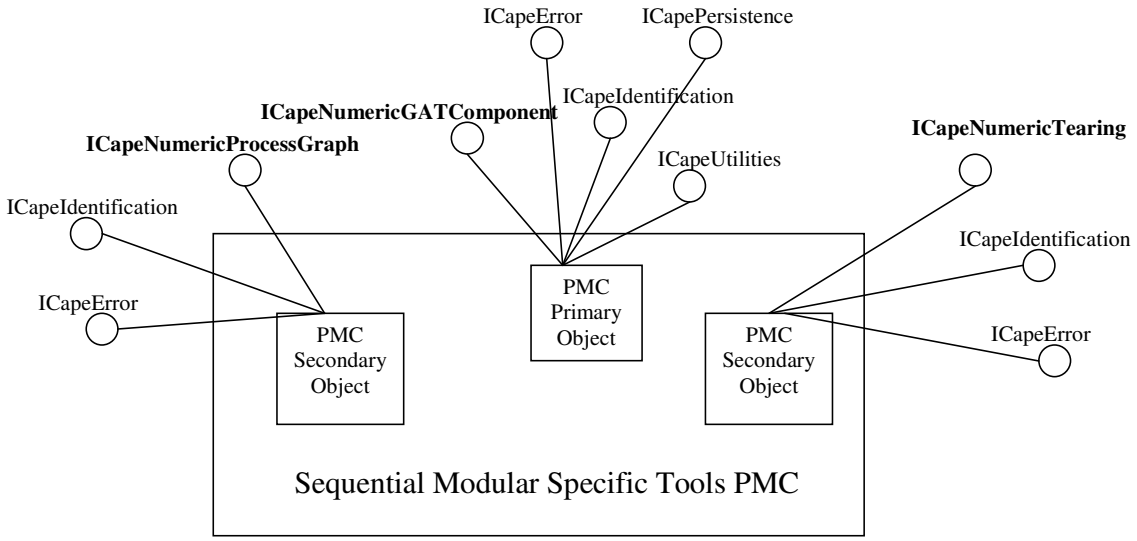


Figure 19 SMST Component Diagram

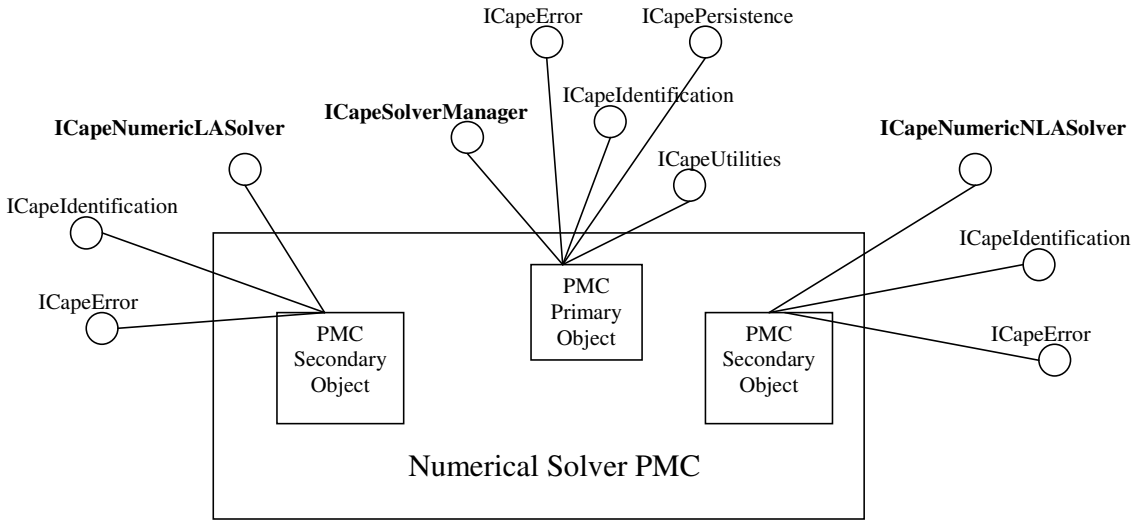


Figure 20 Numerical Solver Component Diagram

9.3 General idea

9.3.1 Needs for CO common interfaces

The *Common Interfaces* are a collection of interfaces that support basic functions and are always independent of *Business/COSE Interfaces*. Within CAPE-OPEN there have been currently few opportunities and initiatives to reuse design concepts across the business topics. In some instances this design reuse could be facilitated to provide one means of achieving consistency across the deliverables of the standard (i.e. the interface specifications). In addition to design reuse, it may be possible to go further and produce implementations of these designs, which are also reusable across the CO component development.

9.3.2 Recommendations to the intended audience

The *Common Interface*s specifications are aimed at designers of CO interfaces and developers of CO components.

DESIGNERS OF CO INTERFACES

They design the *Business/COSE Interfaces* belonging to the CO standard and write the corresponding open interface specification document.

Methods & Tools group requires *Common Interfaces* to be part of future *Business/COSE Interfaces* specification if this specification needs functionalities which can be supplied by already existing *Common Interfaces*. In the case this specification requires further functionalities than the ones provided by *Common Interfaces*, the Methods & Tools group will consider enhancing the *Common Interfaces*.

The designers of a CO interface specification have to use the *Common Interfaces*. Thus the designers have to integrate the existing *Common Interfaces* within their design. The way to include them is described in the *Common Interfaces* specification documents. Indeed each specification document show clearly how to integrate these common functionalities within any design of *Business/COSE Interfaces* . For example the Error Common Interface describes how any *CO object* handles the CO errors. Therefore any *Business/COSE*

Interfaces have to be compliant with these recommendations. The designer should illustrate the dependencies between his specification and the *Common Interfaces*; basically for instance through the UML model by drawing specific diagrams.

COSE Interface

PME objects provide the *Common Interfaces* functionalities as explained in the section 9.1.

Business Interface

The designers of CO interfaces have to clearly identify the interfaces as primary or secondary. With respect to this kind, the resulting *PMC objects* provide the *Common Interfaces* functionalities as explained in the section 9.2.

DEVELOPERS OF CO COMPONENTS

They develop applications/components which are compliant with the CO standard.

The *Common Interfaces* are general purpose interfaces that are mandatory for developing CO-based components. The CO component developer has to implement not only the *Business/COSE Interfaces* but also the *Common Interfaces* which are related. For instance the CO Unit developer has to implement the Identification interface since the Unit interface specification requires the use of the Identification interface.

9.3.3 General design principles

- The *Common Interface*s are built in the same manner than any *Business/COSE Interface* specification. Therefore the content and the syntax used to specify the *Common Interface*s are similar to the ones used to specify any *Business/COSE Interface* specification. Each *Common Interface* is specified by a separate document. So as specified by the Methods & Tools group at present, the Template for Interface Specification Document is the reference document. A priori this the Methods & Tools group that is in charge of producing the *Common Interfaces*.

- The textual requirements and the UML model should make no reference to any other *Business/COSE Interfaces*, since the *Common Interface*s are of general purpose. However, use cases and diagrams can be added as concrete examples of how *Common Interface*s are used by some *Business/COSE Interfaces* and corresponding CO components. Basically the next figure shows the dependency between the *COSE/Business Interfaces* and *Common Interfaces*.

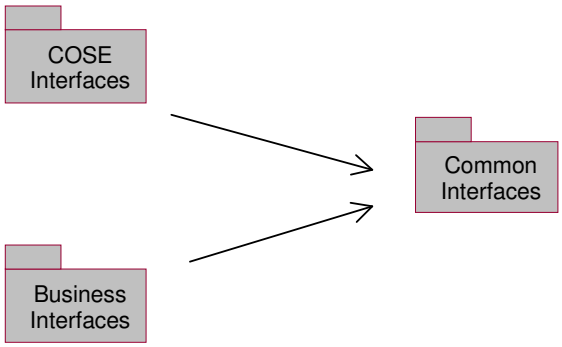


Figure 21 Dependency Relations

- The design of *Common Interface*s leads to interfaces which a priori are integrated directly within the design of any *Business/COSE Interfaces*. It is only a factorisation of general interface, similar to famous design patterns. That means that there is no client/server relation between *Business Interfaces* and *Common Interface*s (from a CO point of view since the proprietary implementation can always distribute subsets of a *PMC*). The resulting CO component will implement not only *Business/COSE Interfaces* but also as required *Common Interfaces*. For instance, the Unit *PMC* will provide the implementation of Unit and Identification interfaces. The interface of Identification *Common Interface* is designed in order to provide interfaces to the *PME* and not to the *PMC* itself.

- The designers of *Business/COSE Interfaces* have to integrate the *Common Interface*(s) in their design as recommended in this part. Each *Common Interface* specification document specifies the integration design to respect.

9.3.4 Versioning aspect

The *Implementation Specifications* under the IDL form is represented by a single library; one for the COM platform and one for the CORBA platform. One version number corresponds to the whole library.

The *Implementation Specifications* enclose the *Common Interface*s. The *Common Interface*s belong to the standard versioning system and so don't have any specific version number. Only a version number for internal use is applied. The designers of the CO interfaces could refer to it but that has no interest for developers of CO components.

The CO components only need to be compliant with a specific version of *Implementation Specification*, for instance version 0.9.3 using the CAPE-OPENv0.9.3.idl which involves some *Common Interface*s.

9.3.5 Associated documents

The CO *Common Interface*s involve the following documents:

- Open Interface Specification: Identification Common Interface;
- Open Interface Specification: Parameter Common Interface;
- Error Handling Strategy: Error Common Interface;
- Open Interface Specification: Utilities Common Interface;
- Open Interface Specification: Persistence Common Interface;
- Open Interface Specification: Collection Common Interface;

9.4 Summary of key features

9.4.1 Error common interface

- This report gives the guidelines to manage the error within any *Business Interface*s and *COSE Interface*s.

- By definition the error is an abnormal termination. It represents a binary status; either there is no error or an error occurs. When a request is made, if this request is successful it raises no error otherwise it raises an error. When an error occurs, the execution is immediately aborted.

- The error strategy is first defined from a conceptual view. Thus the strategy is independent from any architecture, system and implementation language. The result uses the UML notation. Then the error strategy is applied to the COM and CORBA platform.

- This document describes a classification and a hierarchy of potential errors occurring in the CO standard. These errors are common to all the CO interfaces which can easily reuse them.

9.4.2 Identification common interface

- This specification will be used by those CO components that wish to expose its name and description. This information refers to an instance of the component, not to the software class.

- A particular situation in a system may contain several CO components of the same class. The user should be able to assign different names and descriptions to each instance in order to refer to them unambiguously and in a user-friendly way.

- The Unit Operations interface specification has for instance the following requirements: If a flowsheet contains two instances of a Unit Operation of a particular class, the CO Simulator Executive needs to provide the user a textual identifier to distinguish each of the instances. For instance, when the CO Simulator Executive requires to report about an error occurred in one of the Unit Operations.

- The interface contains a straightforward interface called ICapeIdentification.

9.4.3 Parameter common interface

- This specification will be used by those CO components that wish to expose its name and description. This information refers to an instance of the component, not to the software class.

- This specification will be used by those CO components that wish to expose some of its own internal data to its clients, so that the latter may utilise it through standard interface.

- The interface is made up of two different parts, each corresponding to a different client need:

    - The first part is a fixed, static aspect that describes the Parameter, such as a type, name, description, dimensionality and so on. This is proposed to be used to assist the human users in deciding what value to give to the Parameter.

    - The second part deals with value of the Parameter itself. It is expected that the parameter values will change quite frequently both within and outside of the Component that needs it.

Additionally, several parameters of a system may share the same parameter description.

9.4.4 Collection common interface

- The aim of the Collection interface is to give a CO component the possibility to expose a list of objects to any client of the component. The client will not be able to modify the collection, i.e. removing, replacing or adding elements. However, since the client will have access to any CO interface exposed by the items of the collection, it will be able to modify the state of any element.

- CO Collections don't allow exposing basic types such as numerical values or strings. Indeed, using CapeArrays is more convenient here.

9.4.5 Utilities common interface

- This interface represents a holdall concept. That allows to gather many basic functionalities within a single *Common Interface* specification. Of course this design choice is convenient because the services that are integrated in the "utilities" object are straightforward, only apply to *PMC primary object* and does not need to be reuse outside this holdall.

- This interface allows any *PME* to manage simulation context, to collect parameters and to edit the *PMC*.

9.4.6 Persistence common interface

- Most simulation environments allow the possibility to store at any moment the state of a simulation case, in order to be able to restore it at any time in the future. In the CO (distributed) architecture, where different pieces of the simulation may be implemented by different vendors, there must be a standard mechanism to provide this feature.

10. Annexe: Template for interface specifications documents

This section delivered as a separate document called *Template for Interface Specification Documents* in the MS Word format (.doc) and in MS Word model format (.dot).

11. Annexe: COM-CORBA bridging

11.1 COM-CORBA bridging

This subject was almost completely treated during the second six months period. We present here a preliminary study of the strength of the business case for interworking technology within the CAPE-OPEN community. We present also the possible bridging techniques that have been studied by the group. Finally, we will present the bridging approach chosen for the prototype. Technical detail on the prototype and the COM-CORBA data conversion will also be given. The only remaining part for the third six month period is the complete integration test of the bridge with Aspen+ and Hysys. There have been some successful experiments demonstrating the correctness of the approach taken. A final test will follow.

11.1.1 Requirements/Business case for COM/CORBA bridging

A questionnaire was sent to all project partners. As of Jan. 28th, 2000, only seven answers were received, and even one of them mostly saying "I don't know". From the others we can depict what the current situation seems to be.

The interoperability on COM and CORBA components is meaningful only for those ones who have a at least a minimal understanding. It is important as far as some commercial or operational products (COSEs or components) are solely available in one flavour (COM or CORBA). This can be in the form of integrating legacy components in an environment, or of mixing "best-of-class" commercial components.

The business value is not easy to assess. Benefits are expressed in terms of "better and faster" rather than using euros. This question of business value was raised because we were wondering whether the bridging was worth the effort. We will have to go on armed with our belief, supported by some statements from users, that this can bring benefits in the medium and long term. It will be interesting to convert the qualitative estimates into quantitative ones when the effort is completed.

The most frequent use will be with a small number of components from one flavour (e.g. one CORBA component) being accessed from a COSE (e.g. a COM-based COSE). This will run on the company network using several (two?) machines.

There are no conclusions on other middleware because only Steven Groot Wassink replied to this one with something else than "don't know".

In summary, there seems to be some value in bridging COM and CORBA components but real benefits needs to be demonstrated: hopefully the example with the IK-CAPE thermo will give ideas to the partners.

The next part presents technical aspects of bridging and summarises what the group plans to do.

11.1.2 Possible bridging mechanisms

BACKGROUND

An early decision in CAPE-OPEN was to provide all interface specifications in *both* OMG's CORBA IDL and Microsoft's COM IDL (or more strictly in the form of Automation interfaces). These will be referred to as 'CIDL' and 'MIDL' respectively hereafter. The original concept was for a "general" CAPE-OPEN interface definition language to provide the primary version of each interface, with the CORBA and Automation versions being derived from this. This was achieved to some extent in the final documents, in that the interfaces were detailed along with their semantics in a series of tables, before the MIDL and CIDL versions were presented. In practice, however, the development effort inevitably focussed on the particular middleware favoured by the team concerned: for Unit this was Automation, while for Solvers and GAT it

was CORBA. The only group of interfaces implemented for both middlewares was Thermo. In particular, prototypes were largely produced only for one choice of middleware in each case.

However, a CORBA IDL version of the Thermo interface was developed, and a prototype IKCAPE thermo server created in accordance with it. This illustrates how the need for Automation/CORBA interworking is likely to arise. A simulator executive designed to provide Automation "sockets" for a CO thermo package will require some intervening software in order to access this CORBA version. Conversely, an Executive with CORBA sockets would require CORBA/Automation interworking to access (for example) Aspen's Thermo package. In general, it is probably unrealistic to expect every software provider to create two versions of their product, one for each type of middleware. Because there are two implementations of the Thermo interfaces in both COM and CORBA these interfaces were chosen to demonstrate COM-CORBA bridging. The prototype of the bridge will use the native CORBA IK-CAPE thermo components and will enable the use of these components in the COM based simualtor executives Aspen+ and Hysys.

The problem of different middleware approaches used in one context is of course not unique to CAPE-OPEN framework. A considerable effort has been undertaken by the OMG consortium to define a standard for interworking of COM (or Automation) components with CORBA. The result is defined in the CORBA 2.3 specification document, available as http://www.omg.org/corba/corbaiiop.html, chapters 17 and 19. These define standard mappings from interfaces defined in Automation or CORBA to their counterparts in the other language. The OMG's objective is to encourage software companies (probably the ones currently providing ORBs) to create generic bridging software supporting this standard. Later we will present some more detail on existing bridging solutions. Thus, in the next section we present a methodology allowing immediate testing of the interworking concept, before discussing these longer term issues.

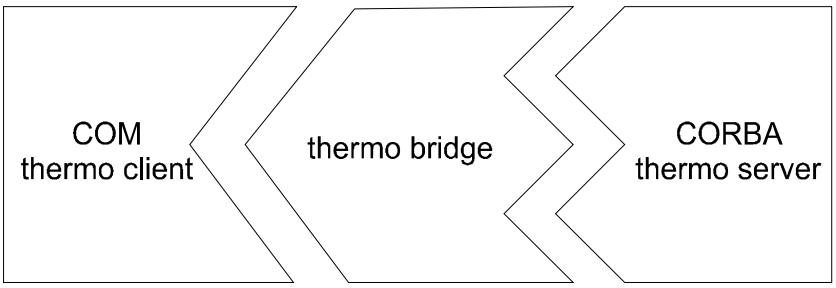


Figure 22 Application of a custom bridge

CUSTOM BRIDGING

In the OMG's terminology, our short term aim is to create a 'custom bridge' between the Automation Thermo socket and the CORBA Thermo plug. This will be a specially written software component which exposes an Automation plug (for example it will be instantiated using the COM mechanisms) but provides a CORBA socket to the IK-CAPE component (for example it might access the server via a CORBA name service). This is illustrated in Figure 22.

The M&T Group has created a custom bridge as described above within the last months as a proof of concept. Using this bridge it will be possible to access *any* CORBA Thermo component from *any* Automation Thermo client — provided of course both conform to the existing IDL definitions that were used for the two prototypes. The bridge implemented in the prototype is a one-way bridge capable of making a native CORBA server available to a COM based client process as depicted in Figure 3. But for some technical reasons explained below the prototype is internally a two-way bridge. Therefore, using the prototype for making COM components available to the CORBA world will take only little effort.

The conformity of MIDL and CIDL has been ensured by applying some minor corrections to the CORBA IDL. The first official releases of the COM type library and the CORBA IDL-file are compliant with regard

to bridging. Therefore, this approach has the benefit of requiring no changes to the MIDL or CIDL versions of the interfaces, or to the prototype codes already developed in accordance with these. It is thus the most conservative approach in terms of working with the end products of the CAPE OPEN project. However, adoption of this approach on a wider scale would imply the production and maintenance of a significant number of such bridges because we need one custom bridge for each interface. Additionally, every change of the standard interfaces implies a change in the according bridge. But this maintenance problem is not as bad as it may sound because the effort needed to adopt a bridge to a change in the interfaces is very small. The reasons for this will become clear when we present the technical detail of the prototype.

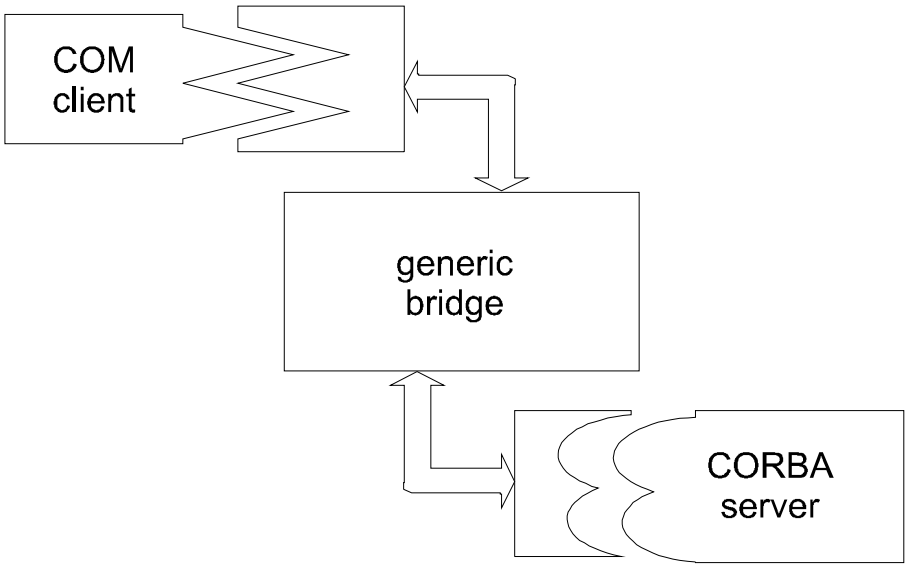


Figure 23 Application of a generic bridge

Nevertheless, a longer term aim be for the two versions of the interfaces to relate to each other in conformity with the OMG's interworking mapping referred to above, which would allow the use of generic bridging tools rather than custom bridges. See Figure 23 for an illustration of this concept. This could be accomplished with the implementation of the interface- and analysis repository located in the CO-LaN implementation. One of its goals is to provide automated generation of MIDL and CIDL from the UML definitions stored in these repositories. This automated IDL generation will be compliant to the OMGs interworking specification. But also the custom bridge tries to take this specification in account as far as possible. We will come back to this later.

The idea of using a generic bridging product as a short term solution was also dropped for some additional reasons. Some commercial bridging products are available, such ase COMet from IONA, the DAIS suite, or a bridge intergrated in the VisiBroker from Inprise. But these products are still at an early stage of development which could bear one or more of the following problems:

- Unstable run-time behaviour
- Most products are only one-way bridges therefore not offering additional functionality to the custom bridge
- Compliance to the OMG bridging specification is questionable which possibly means problems when change the bridging product.

Additionally, these products are not free so everyone wanting to use both middlewares simultaneously in a simulator executive would have to pay extra money. On the other hand the custom bridges could be made freely available on the CO-LaN web-site.

One might ask, since the CIDL and MIDL versions were developed from the same design, why might they fail to satisfy this mapping? This is mainly a question of details arising from decisions on the handling of particular issues such as exceptions, where the OMG's recommendations differ from our own. For example, the current MIDL specification for the ThermoSystem interface is as follows:

```
#ifndef _THERMOSYSTEM_IDL_
#define _THERMOSYSTEM_IDL_
// Provide an interface for Thermo System
// Definition of the Thermo System configuration
import "oaidl.idl";
import "ocidl.idl";

// Include GUIDs
#include "COGuids.idl"

// Fundamental types
#include "Fundamental.idl"

// Material Template and Material Object
#include "Cose.idl"

// ICapeThermoSystem interface
[
       object,
       uuid(ICapeThermoSystem_IID),
       dual,
       helpstring("ICapeThermoSystem Interface"),
       pointer_default(unique)
]
interface ICapeThermoSystem : IDispatch
{
       // Get the list of available property packages
       [id(1), helpstring("method GetPropertyPackages")]
       HRESULT GetPropertyPackages([out, retval] CapeArrayString *propPackageList);

       // Resolve a particular property package
       [id(2), helpstring("method ResolvePropertyPackage")]
       HRESULT ResolvePropertyPackage(
                            [in] CapeString propertyPackage,
                            [out, retval] CapeInterface *propPackObject);
};
#endif //_THERMOSYSTEM_IDL_
```

while the CIDL version is this:

```
#include "base.idl"
#include "cose.idl"

#ifndef THERMO_SYSTEM_IDL
#define THERMO_SYSTEM_IDL

module ThermoSystem {

  interface ICapeThermoPropertyPackage;
  interface ICapeThermoSystem;

  /* Sequence definitions */
  typedef sequence<ICapeThermoPropertyPackage>
                                  CapeThermoPropertyPackageSequence;

  typedef sequence<ICapeThermoSystem>
                                   CapeThermoSystemSequence;


  /* Interface definitions */

  interface ICapeThermoSystem : Cape::ICapeIdentification {
```

```
    Cape::CapeStringSequence GetPropertyPackages();
      // returns a name for all available Property Packages

    ICapeThermoPropertyPackage
    ResolvePropertyPackage(in Cape::CapeString propPkg)
      raises (Cose::CapeThermoUnknownIdentifierException);

      // This method returns Property Package Interface pointer for
      // given Property Package. Exception indicates that package is
       //not available
  };
```

A few points illustrating the details involved in a compliant mapping (with page references to the OMG document for CORBA 2.3) are:

- Due to the presence of a module definition in the CIDL, the OMG mapping would require the MIDL interface to be renamed DIThermoSystem_ICapeThermoSystem.

- Single inheritance of interfaces should be the same in both MIDL and CIDL: thus DIThermoSystem_ICapeThermoSystem should inherit from DICape_ICape-Identification (which might in turn inherit from IDispatch).

- The MIDL operations (GetPropertyPackages and ResolvePropertyPackage) should both have an additional argument marked as [optional, out] to return exception information.

Clearly, there is some effort — and potentially a moderate loss of readability — involved in conforming to these standards, but the potential saving of effort if generic bridging tools can be used appears worthwhile. Thus another area of our activity in the next few months will be concerned with modifying the CIDL and MIDL Thermo specifications[5] so that they become compliant mappings of each other. Figure 24 illustrates the point: here a vertical line denotes a compliant mapping. Thermo is the currently defined MIDL interface, and Cust(Thermo) the currently defined CIDL version. Thermo' represents a modified version of the MIDL interface designed to satisfy the mapping: OMG(Thermo') is the compliantly mapped CIDL version.

[5] The examples given above were presented from the point of view of changing only the MIDL material: however in practice, it will probably be appropriate to make changes to both in order to reach the best compromise between the styles of the two types of middleware. The aim should be to minimise the combined impact of these changes on the effort of maintaining both the Automation and CORBA software.

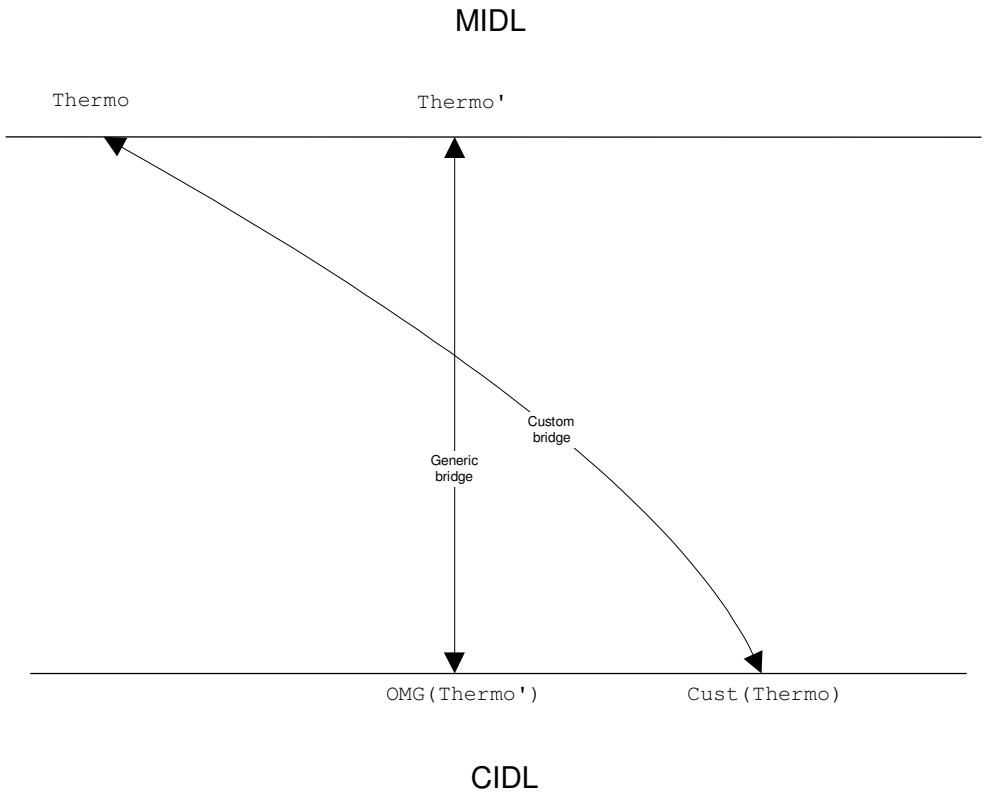


Figure 24 Illustration of different mapping strategies

11.2 The bridging prototype

In this section we present some conceptual and technical background information about the bridging prototype. The prototype that was implemented during the last months is a complete custom bridge for the thermo interfaces in the sense of what was shown in the last section. Its core is based on the IK-CAPE thermo package which was wrapped and made CAPE-OPEN compliant in the CAPE-OPEN project. It was chosen for the prototype testing because thermo seemed to be the simplest solution for testing the bridging. The thermo interfaces can be tested quite standalone and it seems to be interesting to use the IK-CAPE thermo within the commercial CAPE-OPEN compliant simulators Aspen+ and Hysys. Some integration tests with Hysys and the bridged and wrapped IK-CAPE thermo have been done and were basically successful. Further testing and a complete plug-and-play integration for both simulators is planned and is aimed to be demonstrated in November 2000.

Although the prototype is restricted to the thermo interfaces and capable of bridging CORBA to COM only it we be no large effort to extend it to the remaining CAPE-OPEN interfaces and the direction COM to CORBA. The reasons for this will become clearer in the following sections.

11.2.1 Technical background

We will now describe in more detail which functionality is included in the bridging prototype and what tools were used for its implementation. Furthermore we will present the problems that had to be addressed in the implementation phase.

As the agreed by the Methods and Tools Group the scenario for the bridging prototype is based on the IK-CAPE CORBA thermo server and properties package. This package which was originally written in FORTRAN and is available as a Solaris library resp. Windows DLL was made CAPE-OPEN compliant by wrapping it with a CORBA shell. It was decided to make this wrapped IK-CAPE available to the COM world via a COM-CORBA bridge. This bridge should then be used to employ the wrapped and bridged IK-CAPE in Aspen+ and Hysys. As these systems are Windows based and the IK-CAPE CORBA server is Solaris based we have the following scenario:

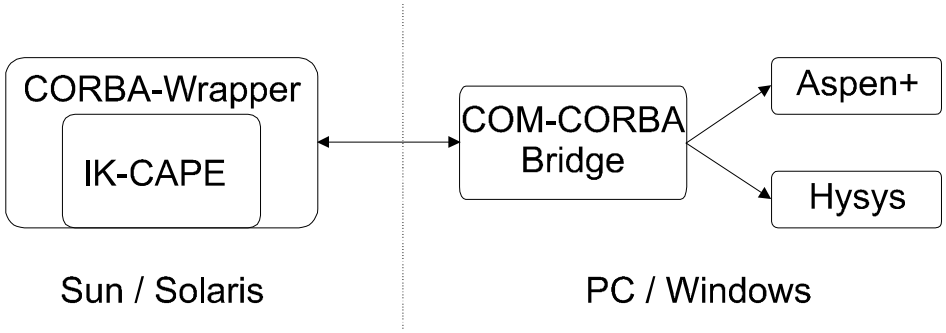


Figure 25 General Bridging Scenario

In this scenario it is quite natural that the bridge resides on the Windows system since COM is a native Windows technology. Although there are implementations of COM and DCOM for Unix/Solaris based systems we have decided to build the bridge on Windows itself. This has several reasons: First, building the bridge on Unix require an additional COM implementation on Unix which is expensive. As we want to use a much free software for the bridge as we can this is not a good idea because COM is already shipped with Windows at no extra cost. Second, as the bridge is designed to be usable for any CORBA CAPE-OPEN thermo system regardless whether it resides on Windows, Linux, Solaris or whatever we need only one bridge for the Windows system and not one for each operating system.

For implementation we decided to use Microsoft Visual C++ 6.0 which offers quite good support for programming COM based applications via the ATL-Library. For the connection to the CORBA parts we used OmniOrb 2.7.3 which was also used for wrapping the IK-CAPE package on the Solaris side. But both products can be changed as not too much code is specific to these systems. Especially for CORBO it is no problem to plug CORBA servers that are not based on OmniOrb into the bridge since IIOP, the communication protocol of CORBA, is standardised. Some ORB integration tests which were carried out in the early phases of Global CAPE-OPEN have shown that this really the case (with some difficulties). Before we come to the technical details of the implementations we will present some general properties of the bridging prototype.

11.2.2 Features of the Bridging Prototype

As stated above the bridging prototype implements a custom one-way-bridge which enables us to make implementations of CORBA thermo interfaces available to the COM world but not vice versa. Additionally the prototype is restricted to the thermo interfaces only. These decisions were made because the bridge is supposed be a proof of concept only. As we will see later the concepts used in the prototype can be easily extended to the other CAPE-OPEN interfaces. We will also see that the implementation of a bridge working the other way namely making COM components available to CORBA is also conceptually captured in the prototype. Some features necessary for this have already implemented in the prototype. As there are certain callback mechanisms in the interactions between a Material Object and a Properties Package these features were implemented to make the one-way bridge work.

Another important restriction of the bridge is that it is a static one. We can distinct two different kinds of bridges: static and dynamic bridges. Dynamic bridges are server processes which can be used to create COM objects the are connected CORBA objects at runtime. If you need COM access to a CORBA object you would have to render a CORBA reference to the server process which then would create end expose an appropriate COM object. This object would contain the CORBA reference handed over to the server before and use it to access the CORBA object. Such bridges can be used to create COM objects dynamically for arbitrary CORBA objects at runtime. Depending on whether the bridge is generic or custom CORBA objects with arbitrary interfaces can be plugged into the server. Some of the generic bridges mentioned above work that way, i.e. are dynamic bridges.

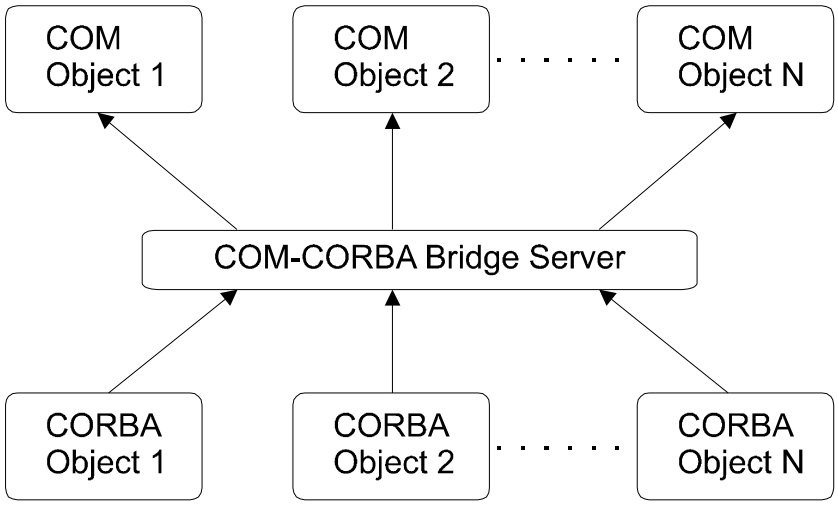


Figure 26 Dynamic Bridge

Static bridges are quite similar to wrappers. The only difference to the kind of wrapper used for making IK-CAPE CAPE-OPEN compliant is that the code that is wrapped is not necessarily on the same machine. As we deal with CORBA objects the CORBA implementation to be provided with a COM wrapper can reside somewhere in a network and is accessed trough the CORBA reference. In contrast to the dynamic approach there is no server process which can generate new COM objects for existing CORBA objects at runtime. The connection between COM and CORBA is established at compile time directly via a set of classes implementing the bridging. These classes can be reused for different CORBA objects if they have the same interface.

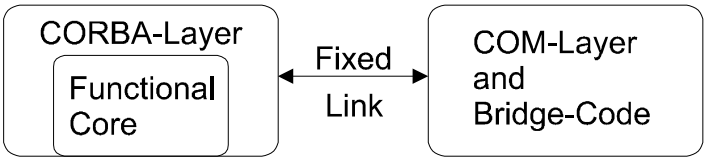


Figure 27 Static Bridge

Static bridges are simpler to implement because the server process managing the run-time generation of new COM-CORBA connections has not to be implemented. But in their core both strategies are similar because all data and calling conversions are the same. There is only more effort to be put in the handling of the objects. Therefore, in this prototype the static and custom approaches were taken. The serve as a proof of concept and can be extended to the other approaches by implementing the necessary features. But the very core of the bridge will stay the same.

As another restriction the prototype does not cover any error handling. At the time of its implementation no error handling strategy was defined. Although there were some exceptions defined in the CORBA thermo interfaces these were completely ignored. Error handling and a translation between the according COM and

CORBA concepts will be one of the future issues for COM-CORBA bridging in CAPE-OPEN. We will now present the main technical problems that had to be solved for implementing the bridge and after that show how these problems were solved.

11.2.3 Technical Challenges

As the COM and CORBA interfaces were derived more or less independently from the UML specifications they are not in line with the OMG's specifications for COM-CORBA mappings. Therefore, the mapping specifications of the OMG have only limited influence on the implementation of our bridge. We have regarded the existing interfaces as fixed and have developed our own mappings where needed. In principle this no problem we just loose the compliance to the OMG's mapping. However most basic data types which could be mapped in compliance to the OMG's definitions for COM/Automation to CORBA interoperability. These basic types (which are aliases for the basic CAPE-OPEN types such as CapeLong) are:

CAPE-OPEN Type	**COM IDL Data Type**	**CORBA IDL Data Type**
`CapeLong`	`long`	`long`
`CapeDouble`	`double`	`double`
`CapeString`	`BSTR`	`string`
`ICapeXXX`	`LPDISPATCH`	`CORBA object reference`
`CapeVariant`	`VARIANT`	`any`

The basic data types which are not mapped in a OMG compliant way are:

CAPE-OPEN Type	**COM IDL Data Type**	**CORBA IDL Data Type**
CapeBoolean	VARIANT_BOOL (should be boolean)	boolean
CapeDate	DATE	string (should be double)

In compliance to the OMG's defintions all CORBA sequences for the basic types and for the interfaces were mapped the Safearrays. How the actual mappings were implemented will be presented in the following section.

In order to create the bridging prototype custom bridges for the following CAPE-OPEN interfaces had to be built:

- ICapeThermoPropertyPackage
- ICapeThermoSystem
- ICapeThermoMaterialTemplate
- ICapeThermoMaterialObject

The first two interfaces encapsulate the core functionality of the IK-CAPE package and the latter ones were needed to test the prototype and to access the properties and calculating methods in the properties package. This strong connection among the interfaces has caused has caused some problems and lead to inefficiencies in implementation. This is due to the fact that for accessing and storing data for a calculation callbacks from a properties package to a material object are needed. The problem was already addressed in the CAPE-OPEN internal paper " A critical assessment of CAPE-OPEN Interfaces from a user's view point". We will now

describe this problem in more detail using a scenario for a call to a properties package calcProp method for calculating an enthalpy.

If we want to calculate an enthalpy in a liquid mixture via an MaterialObject and a PropertyPackage the following calls are made:

```
mo->setIndependentVar(...);   // set temperature and pressure
proplist[0] = "enthalpy";
phases[0] = "liquid";
mo->calcProp(proplist,phases,"Mixture")
```

Here we can see that we need the MaterialObject Bridge for testing the PropertyPackage which is called within the calcProp method of the MaterialObject. The MaterialTemplate is needed for creating the MaterialObject. This situation causes no problem concerning the bridge. If we have CORBA MaterialObjects we just need an appropriate simple bridge. The problems arise when we take a look into the calcProp Method. Within this call a call to the calcProp Method of the PropertyPackage is made with the MaterialObject as Parameter:

```
pp->calcProp(mo,proplist,phases,"Mixture");
```

The situation is the same when the calculation is called directly from outside. The problem is that the MaterialObject is needed as parameter for the PropertyPackage calcProp implementation for retrieving or storing data. As a result in this case a simple bridge for the COM-CORBA bridge for the PropertyPackage is not sufficient. Let us consider the following situation: A native CORBA PropertyPackage is present as it is the case in our IK-CAPE scenario. Therefore, we have to implement a bridge for this package making it available as COM object. If we want to perform a calculation with the COM object we need a COM MaterialObject to exchange data as seen above. But in the COM implementation of the PropertyPackage bridge a CORBA call to native CORBA implementation of the PropertyPackage is issued. This call requires a CORBA MaterialObject as Parameter. This would be no problem if the COM MaterialObject put as parameter into the bridge were also a native CORBA object with some bridging code around it. But in general this will not be the case. Therefore, we have the situation of having to plug the native COM MaterialObject into the CORBA call to the PropertyPackage. As a result we need a CORBA wrapper for the COM MaterialObject.

Therefore our bridging prototype contains not only code for making CORBA objects available as COM object but vice versa as well. This is why we stated earlier that in fact we do not only have one-way bridging code but two-way code. However, the code is available for both directions only for the MaterialObject but as we will see later using it for other interfaces is more or less a matter of cut-and-paste. In the next section we will give a brief overview of the implementation itself.

11.2.4 Overview of the Implementation

As MS Visual C++ offers quite good support for writing ATL (Active Template Library) based Automation/COM applications via some wizards only the core functionality of the bridges had to be implemented manually. Only the method implementations of the server skeletons had to be filled and the data conversion between COM and CORBA had to be done. Therefore the prototype basically consists of the following main modules:

- Data conversion module which implements the mappings for the basic data types and sequences as presented in Section 11.2.3.
- CORBA-to-COM bridge for the interfaces ICapeThermoSystem, ICapeThermoPropertyPackage, ICapeThermoMaterialTemplate, and ICapeThermoMaterialObject

- COM-to-CORBA bridge for the ICapeThermoMaterialObject interface needed for the situation depicted in the last section.

We will now describe the modules in more detail.

DATA CONVERSION

As we have seen before the data types (basic and complex) are not the same for COM and CORBA. In order to convert parameters and results of calculations a set of data conversion routines has been implemented. The routines are the very core of the bridge and are used in almost every method of the bridges. These conversion routines are not specific to the thermo interfaces and can be used in custom bridges for other interfaces without any change. They can also be used in more advanced bridges like dynamic and/or generic bridges. The conversion module contains the following routines:

```
BSTR String_2_BSTR ( const char * const CORBA_String);
Cape::CapeString BSTR_2_String (const BSTR COM_String);
Cape::CapeBoolean VTBOOL_2_boolean(const CapeBoolean COM_Bool);
CapeBoolean boolean_2_VTBOOL (const Cape::CapeBoolean CORBA_Bool);

void COM_CapeVariant_to_CORBA_CapeStringSequence
   (Cape::CapeStringSequence * CORBA_StringSequence,
    const CapeArrayString COM_StringVariant);
void CORBA_CapeStringSequence_to_COM_CapeVariant
   (const Cape::CapeStringSequence CORBA_StringSequence,
    CapeArrayString*  Com_StringVariant);
void CORBA_CapeLongSequence_to_COM_CapeVariant
   (const Cape::CapeLongSequence CORBA_LongSequence,
    CapeVariant*  COM_LongVariant);
void COM_CapeVariant_to_CORBA_CapeLongSequence
   (Cape::CapeLongSequence* CORBA_LongSequence,
    const CapeVariant COM_LongVariant);
void CORBA_CapeDoubleSequence_to_COM_CapeVariant
   (const Cape::CapeDoubleSequence CORBA_DoubleSequence,
    CapeVariant*  COM_DoubleVariant);
void COM_CapeVariant_to_CORBA_CapeDoubleSequence
   (Cape::CapeDoubleSequence * CORBA_DoubleSequence,
    const CapeVariant COM_DoubleVariant);
void CORBA_CapeBooleanSequence_to_COM_CapeVariant
   (const Cape::CapeBooleanSequence CORBA_BooleanSequence,
    CapeVariant*  COM_BooleanVariant);
void COM_CapeVariant_to_CORBA_CapeBooleanSequence
   (Cape::CapeBooleanSequence * CORBA_BooleanSequence,
    const CapeVariant COM_BooleanVariant);
```

The first four routines convert the basic types boolean and string from COM to CORBA and vice versa. Their implementation is quite simple and contains only a few lines of code. Nevertheless implementing the conversion required a lot of work for finding out which routines had to be called. This was a general problem when writing the conversion routines. The routines as such were quite straightforward but finding out which API routines to use for it was very complicated. As low level C++ COM programming is far from being simple this has required some intensive investigation on how COM and Automation handle their data. Conversion routines for long and double are obviously not needed as both are the same in COM and CORBA. Conversion for object references was not implemented because this is very simple and can be performed directly where needed. Additionally, the code to be written there depends on what kind of object reference (i.e. what interface) you want to handle which would require routines for all interfaces. In the sense of generally applicable conversion routines we considered it not to be relevant to implement.

Conversion for CapeVariant and CapeDate was not implemented as it was not needed in our prototype.

THE BRIDGES

We will now present a short overview over the implementations of the different bridges. As all custom bridges look very similar we will take only three examples demonstrating the basic concepts. We will show

one example for both COM-to-CORBA and CORBA-to-COM bridging and the implementation of the calcProp Method of the PropertyPackage which contains the dynamic conversion from COM to CORBA objects.

All CORBA-to-COM bridges internally hold a CORBA object reference providing access to the native implementation of the object. Therefore, the MaterialObject bridge holds a reference to the CORBA MaterialObject implementation. How this reference is obtained depends on the interface the bridge is designed for. The PropertyPackage, ThermoSystem and MaterialTemplate bridges obtain the CORBA reference in their constructors via the CORBA naming service. Therefore, in this prototype the connection of the bridge to the CORBA object is fixed. It would be no problem to break this up and add a parameter to the constructor containing the CORBA reference. The object reference for the MaterialObject is explicitly set by the MaterialTemplate which creates it. This has some technical reasons which will not be explained here. The situation is the analogous for the COM-to-CORBA bridge for the MaterialObject but here a COM reference is used internally and explicitly set from outside. These internal references are then used to call the native COM or CORBA methods.

Every bridge CORBA-to-COM implements every method for its interface. But as it is designed to handle arbitrary CORBA objects internally it contains no computational logic of its own. The only thing it does is to forward a call made to the COM object (i.e. the bridge object) to the CORBA object. To do so every bridge method follows the same scheme:

(i) Convert data types of COM parameters to CORBA data types using the conversion routines mentioned above.

(ii) Call the CORBA object using the internal reference with the converted parameters (result of step 1)

(iii) Convert the result of the CORBA call back to COM data structures and set the COM out parameters appropriately.

As an example we present the GetComponentConstant method of the MaterialObject.

```
STDMETHODIMP CCapeThermoMaterialObject::GetComponentConstant
    (CapeVariant props, CapeVariant compIds, CapeArrayDouble * propVals) {
  // create CORBA data structures
    Cape::CapeStringSequence CORBAstringSequence_props;
    Cape::CapeStringSequence CORBAstringSequence_compIds;
    Cape::CapeDoubleSequence_var
      CORBAdoubleSequence_results = new Cape::CapeDoubleSequence();
  // convert COM parameters to CORBA
    COM_CapeVariant_to_CORBA_CapeStringSequence
      (&CORBAstringSequence_props,props);
    COM_CapeVariant_to_CORBA_CapeStringSequence
      (&CORBAstringSequence_compIds, compIds);
  // call the CORBA implementation
    try {
      CORBAdoubleSequence_results = pMaterialObject->GetComponentConstant
         (CORBAstringSequence_props,CORBAstringSequence_compIds);
    }catch (CORBA::SystemException& e) {
      check_exception(e);
      return S_FALSE;
    }
  // convert the results back to COM
    CORBA_CapeDoubleSequence_to_COM_CapeVariant
      (CORBAdoubleSequence_results,propVals);
    return S_OK;
}
```

The according function in the COM-to-CORBA bridge follow the same principle but have to do the data conversions in the reverse order. As an example we present the same methods of the COM-to-CORBA bridge:

```
Cape::CapeDoubleSequence * corbaMOwrapper::GetComponentConstant
  (const Cape::CapeStringSequence & props, const Cape::CapeStringSequence & compIds) {

  //create COM data structures and convert the parameters
  //from CORBA to COM
  CapeArrayString * comProps = new CapeArrayString;
  CORBA_CapeStringSequence_to_COM_CapeVariant(props, comProps);
  CapeArrayString * comCompIds = new CapeArrayString;
  CORBA_CapeStringSequence_to_COM_CapeVariant(compIds, comCompIds);

  _variant_t comResults;
  //call the COM implementation
  try {
    comResults = comMO->GetComponentConstant(comProps,comCompIds);
  }catch (_com_error e) {
     cout << e.ErrorMessage();
     return NULL;   }
  //convert the results
  Cape::CapeDoubleSequence * corbaResults =
    new Cape::CapeDoubleSequence();
  COM_CapeVariant_to_CORBA_CapeDoubleSequence(corbaResults,comResults);
  return corbaResults;
}
```

As we can see here the implementation of the bridge is very generic and can be adopted to all other CAPE-OPEN interfaces very easily. This is also an advantage if the interfaces change over time (what they surely will) because adopting the bridges to a new version of the standard will be no large effort.

As the last example of the bridge implementation we will show how the callback problem to the MaterialObject within the calcProp method of the PropertyPackage is handled. This situation is not specific to just this method but will occur every time a complex COM object is passed through the bridge as a parameter to a method of a native CORBA object. Then, the COM object has to be plugged into an COM-to-CORBA bridge (or a CORBA wrapper for the COM object) before the native CORBA method can be called and the object is passed as a parameter. With the bridge the CORBA object can in its implementation access the COM object's implementation and call the method it needs. This procedure looks in case of the calcProp methods of the CORBA-to-COM PropertyPackage bridge as follows:

```
STDMETHODIMP CCapeThermoPropertyPackage::CalcProp
  (CapeInterface materialObject, CapeVariant props, CapeVariant phases,
   CapeString calcType) {
// create CORBA data structures
  Cape::CapeStringSequence _props;
  Cape::CapeStringSequence _phases;
  Cape::CapeString _calcType;
// convert COM parameters to CORBA data structures
  COM_CapeVariant_to_CORBA_CapeStringSequence(&_props ,props);
  COM_CapeVariant_to_CORBA_CapeStringSequence(&_phases, phases);
  _calcType = BSTR_2_String(calcType);
// get COM MaterialObject from COM parameter
  MATOBJWRAPPERLib::ICapeThermoMaterialObjectPtr pCTMO;
  materialObject->QueryInterface(&pCTMO);
// create CORBA wrapper for the COM MaterialObject
  corbaMOwrapper * cmow = new corbaMOwrapper(pCTMO);
  cmow->_obj_is_ready(boa);
  try {
// call CORBA method with bridged COM object. The CORBA CalcProp method
// internally accesses the MO via a CORBA reference
   pPropPack->CalcProp(cmow,_props,_phases,_calcType);
  }catch (CORBA::SystemException& e) {
    check_exception(e);
    return S_FALSE;
  }
// no results have to be converted here. The calcProp results are stored
// internally in the MO
  return S_OK;
}
```

11.3 Conclusion and Further Work

In this paper we have presented an overview over the COM-CORBA bridging prototype and its implementation. Although the implementation looks quite simple and is not too large it was quite complicated to get it running. The main problem is that low level COM programming with C++ is a quite complex task. But the result has shown as a proof of concept that the IK-CAPE package can be used within a COM based environment. Further tests will have to be performed especially for integration in Aspen+ and Hysys. A integration test with Hysys was a partial success and will have to be repeated in the near future.

We have also seen that if there are some resources and interest in the topic the bridge can easily be extended for other CAPE-OPEN interfaces. Another interesting issue could be an alternative implementation using Visual Basic. As there is a new CORBA ORB for Visual Basic which seems after some initial testing quite stable it could be promising to migrate the bridge to visual basic because COM programming is a lot easier there.