CAPE-OPEN

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

Open Interface Specification: Utilities Common Interface

CO LaN

www.colan.org

ARCHIVAL INFORMATION

Filename	Utilities Common Interface.doc
Authors	CO-LaN consortium
Status	Public
Date	August 2003
Version	version 2
Number of pages	29
Versioning	version 2, reviewed by Jean-Pierre Belaud, August 2003
	version 1, Methods & Tools group, November 2001
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	

IMPORTANT NOTICES

SUMMARY

This document describes a Common Interface proposed by the Methods & Tools group: the Utilities Common Interface. The Common Interfaces are interfaces and implementation models for handling concepts that may be required by any *Business interfaces* and *COSE interfaces*.

This interface represents a holdall concept. That allows gathering 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. All these functionalities could be grouped in a single interface, to be called Utilities.

This interface **has to be provided by any PMC primary object**. That allows **any PME to manage simulation context, to collect parameters of PMC and to edit the PMC**.

ACKNOWLEDGEMENTS

CONTENTS

LIST OF FIGURES

1. Introduction

As described in the Methods and Tools Integrated Guidelines, the services provided by a PMC are supplied by different objects within the PMC: a primary object and a set of secondary ones. A set of *Common Services* are defined to be implemented by the primary one in order to provide horizontal requirements, such as persisting the component state. Still, there is a list of unrelated horizontal functionalities that do not fit in any of these Common Interface specifications.

This new interface may act as a integrator of all the CO interfaces exposed by the PMC. For example, to allow the PMC to benefit from CO services provided by the PME (what is technically a **callback**), the latter must provide the former with a reference to itself. Currently, there is no Common or Business interface where we could add the operation that will allow setting the reference (although the unit interface specification evidences the requirement). So, the new Utilities interface will include all those operations that refer to a whole PMC instance, which means that they may have impact on any interface exposed by the PMC.

2. Requirements

These requirements express the need for the CO standard to have a Utilities CAPE-OPEN interface.

2.1 Textual requirements

When a PME requires some kind of functionality, with the help of the CAPE-OPEN categories, the user is able to select and create a CO class which will expose the required CO interfaces. There is the need for the PME to exchange some information with this instance of the PMC. This information consists in a set of simple unrelated functionalities that will be useful for any kind of CAPE-OPEN component, since they will allow maximum integration between clients and servers. All these functionalities can be grouped in a single interface. Some of the functionalities to fulfil consist in exchanging interface references between the PMC and the PME. Instead of adding these properties to each business interfaces, it is much more convenient to add them to a single common interface which refers to the whole PMC.

Furthermore, there is a need for getting parameters, editing and lifecycling.

The interface should fulfil the following requirements:

Parameters:

So far, only Unit Operations can expose their public parameters, through property ICapeUnit.parameters, which returns a collection of parameters. This property allows COSEs to support design specs between two CAPE-OPEN Unit Operations. That means that the CAPE-OPEN interfaces are powerful enough to allow that the design spec of a given Unit Operation (exposed through public parameters) depends on transformations of public parameters exposed by other CAPE-OPEN Unit Operations. If also other components, such as Material Object, would be able to expose public parameters, the functionality the aforementioned described functionality could be extended. Other functionalities would be:

(i) allowing optimizers to use a public variable exposed by any CAPE-OPEN component.

(ii) Allow performing regression on the interaction parameters of a CAPE-OPEN Property Package.

Centralising the property that accesses these collections in a single entry point, helps to clarify the life cycle usage standards for these collections. That means that it will be easier for the PMC clients to know how often they have to check whether the contents of these collections have changed (although the collection object will be valid until the PMC is destroyed). Setting general rules for the usage of these collections makes the business interface specifications more regular and simpler. Obviously, since too general rules may reduce flexibility, PMC specifications might point out exceptions to the general rule. Let’s see how would this affect the particular PMC specifications:

Simulation context:

So far, most of CAPE-OPEN interfaces have been designed to allow a client to access the functionality of a CAPE-OPEN component. Since clients will often be Simulation Environment, CAPE-OPEN components would benefit from using functionality provided by their client, a COSE for instance. These services provided by any PMEs are defined within the Simulation Context COSE Interface specification document. The following interfaces are designed:

(i) **Thermo Material Template Systems**: Theses interface allows a PMC to choose between all the Thermo Material factories supported by the PME. These factories will allow the PMC to create a thermo material object associated to the elected Property Package (which can be CAPE-OPEN or not).

(ii) **Diagnostics**: This interface will allow to integrate seamlessly the diagnostics messages generated by any PMC with the mechanisms supported by the PME to display this information to the user.

(iii) **COSEUtilities**: In the same idea of this specification document, PME has also its own utilities interface in order to gather many basic operations. For instance that allows the PME to supply a list of standardised values.

Edit:

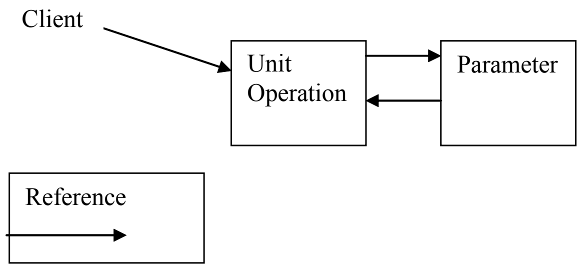
The Edit method defined by the UNIT specification proved to be very useful in order to provide Graphical User Interface (GUI) capabilities highly customized to each type of UNIT implementation. There's no reason why other PMCs could not benefit of this capability. Obviously, when a PMC provides Edit functionality, being able to persist its state is a desired requirements, to prevent the user from having to repeatedly reconfigure the PMC.

LifeCycle:

There is probably no strict necessity to expose directly initialization nor destruction functions, since these should be invoked automatically by the used middleware (COM/CORBA). That is, the initialization could be performed in the constructor of the class and the destroy in its destructor. However, in some cases the client could need to invoke them explicitly. For example, all actions that could fail should be invoked by these methods. If these actions were places in the constructor or destructor, potential failures would cause memory leak, and they would be difficult to track, since it would not be clear if the component has been created/destroyed or not. Examples of cases where they are useful:

(i) **Initialize**: The client might need to initialize a given set of PMC in a specific order, in case that there are dependencies between them. Some PMC may be wrappers to other components, or may need an external file to get initialized. This initialization process may often fail, or the user may even decide to cancel it. Moving these actions from the class constructor to the initialize method allows communicating the client that the construction of the component must be aborted in some cases.

(ii) **Destructors**: The PMC primary object should destroy here all its secondary objects. Relying on the native destructor could cause deadlocks when loop references exist between PMC objects. See in the example diagram below that after the client releases its reference to the Unit Operation, both the Unit and the Parameter are being used by another objects. So, without an explicit terminate method, none of them would be ever terminated.



2.1.1 Future services

Although not currently defined, other functionalities to be defined in the future could be added. For example,

Events:

In case that it is decided not to use COM/CORBA native events, a simple method such as ConsumeEvent (eventType) could fulfill simple requirements such as stopping, pausing or resuming calculations.

2.2 Use cases

2.2.1 Actors

Simulator Executive: The software that controls the simulation,for example determines computational sequence, and calls physical property computations at the appropriate times.

Process Modelling Component (PMC): Such as a Unit Operation or a Property Package.

2.2.2 List of Use Cases

- UC-001: Get List of parameters
- UC-002: Set Simulation Context
- UC-003: Destruction of PMCs
- UC-004: Edit

Use Cases Categories:

- **Parameters Use Cases:** The document 'Open Interface Specification: Parameter Common Interface' contains the list of use cases about the use of parameters. The current document only describes the use case necessary for the Simulator Executive get the reference to the Parameters interface.
- **Simulation Context:** The document 'Simulation Context Interfaces' contains the list of use cases about the use of simulation context interfaces. The current document only describes the use case necessary for the PMC get the reference to the Simulation Context interfaces.
- **PMC LifeCycle:** Use cases about creation and destruction of objects.
- **Edit:** A PMC shows its custom GUI.

Use Cases Priorities:

- **High** Essential functionality. Functionality without which usability or performance might be seriously compromised
- **Low**. Desirable functionality that will improve performance. If this Use Case is not met, usability or acceptance can decrease.

2.2.3 Use Cases Maps

2.2.4 Use Cases

UC-001: GET LIST OF PARAMETERS

Actors: Simulator Executive

Priority: <High>

Classification: <Parameters Use Cases >

Context: The PMC has been created by a Simulator Executive.

Pre-conditions:

The PMC has been correctly initialized

Flow of events:

The PME such as Simulator Executive asks the Parameters Owner to supply its list of Parameters, and this action is fulfilled by the Parameters.

The Simulator Executive then can browse on the list of Parameters to obtain the desired Parameter for e.g. displaying, changing and/or validating its configuration.

Post-conditions:

The PMC has delivered the list of Parameters successfully.

The Simulator Executive succeeds in getting the desired Parameter

Errors:

The PMC fails while delivering the Parameters list

Uses:

Extends:

UC-002: SET SIMULATION CONTEXT

Actors: Simulator Executive

Priority: <High>

Classification: <Simulation Context >

Context: The PMC has been created by a Simulator Executive.

Pre-conditions:

The PMC has been correctly initialised

Flow of events:

The PME such as Simulator Executive conveys the PMC a reference to the former's simulation context.

The PMC can then call back the PME in order to benefit from its exposed services

Post-conditions:

The PMC holds a reference to the simulation context.

Errors:

The PMC fails while accepting the Simulation Executive reference.

Uses:

Extends:

UC-003: DESTRUCTION OF PMCS

Actors: Simulator Executive

Priority: <Low>

Classification: <PMC Lifecycle>

Context: The Simulator Executive does not need the PMC anymore

Pre-conditions:

Flow of events:

The PME such as Simulator Executive requests the PMC to be terminated. The PMC performs any uninitialization task, such as terminating secondary objects.

Post-conditions:

The PMC does not accept any more request after being terminated.

Errors:

Uses:

Extends:

UC-004: EDIT

Actors: Simulator Executive

Priority: <Low>

Classification: <Edit>

Context: The PMC has been created by a Simulator Executive.

Pre-conditions:

Flow of events:

The PME such as Simulator Executive requests the PMC to display it GUI. The user will now interact with the PMC's interface instead of doing it with the simulator's GUI. When the PMC GUI is closed, the simulator must check whether the PMC state (such as number or configuration of parameters or ports) has been modified and integrate these modifications. After that the user will get access back to the simulator's GUI.

Post-conditions:

Errors:

Uses: Extends:

2.3 Sequence diagrams

None.

3. Analysis and Design

3.1 Overview

As described in the introduction, the requirements described in these document all refer to functionalities that may be centralised in a single object: the PMC primary object of any PMC.

3.2 Sequence diagrams

SQ-001 SEQUENCE DIAGRAM

The following basic sequence diagram shows the interactions that are related to the Utilities functionalities between any PME and any PMC.

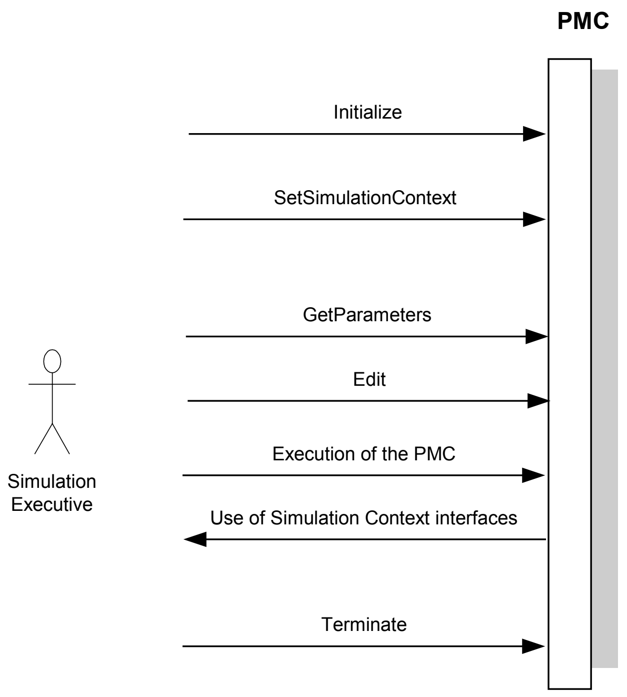


Figure 1 Sequence diagram

3.3 Interface diagrams

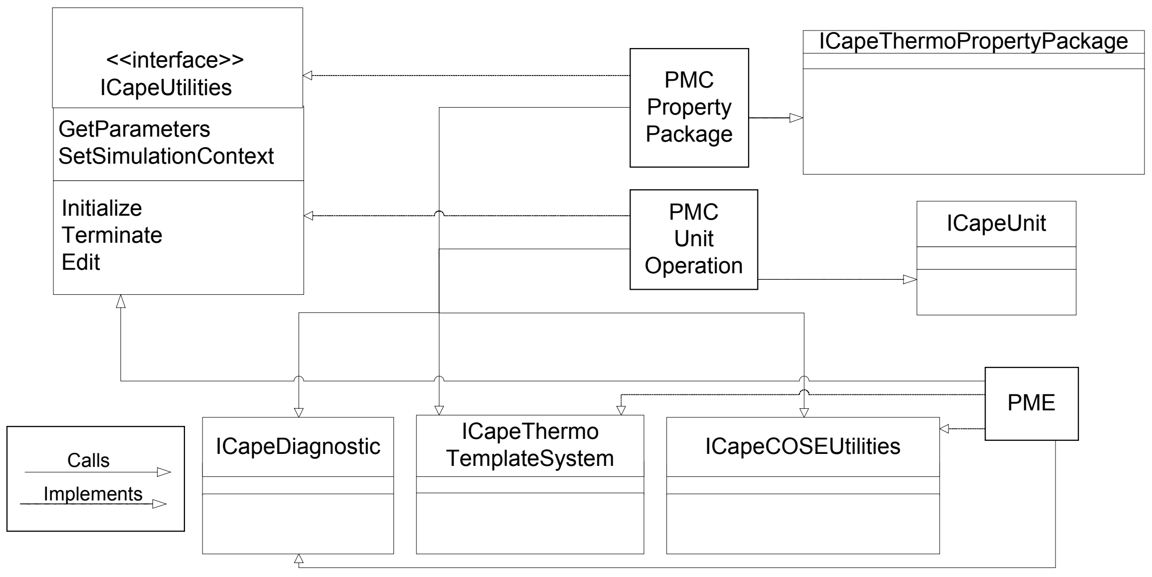


Figure 2 Interface diagram

3.4 State diagrams

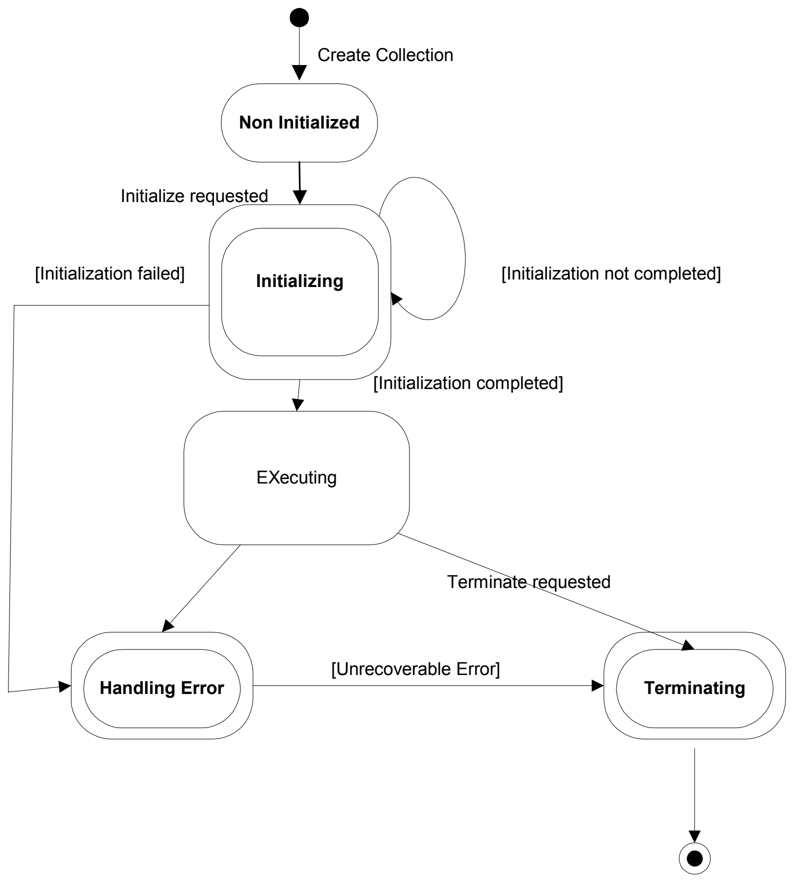


Figure 3 State diagram

3.5 Other diagrams

The following diagram confirms that the Utilities functionalities are only implemented and provided by any PMC primary object. As illustration we show a component diagram for a unit operation. The interfaces in bold belong to *Business Interface* category while the others belong to the *Common Interface* category.

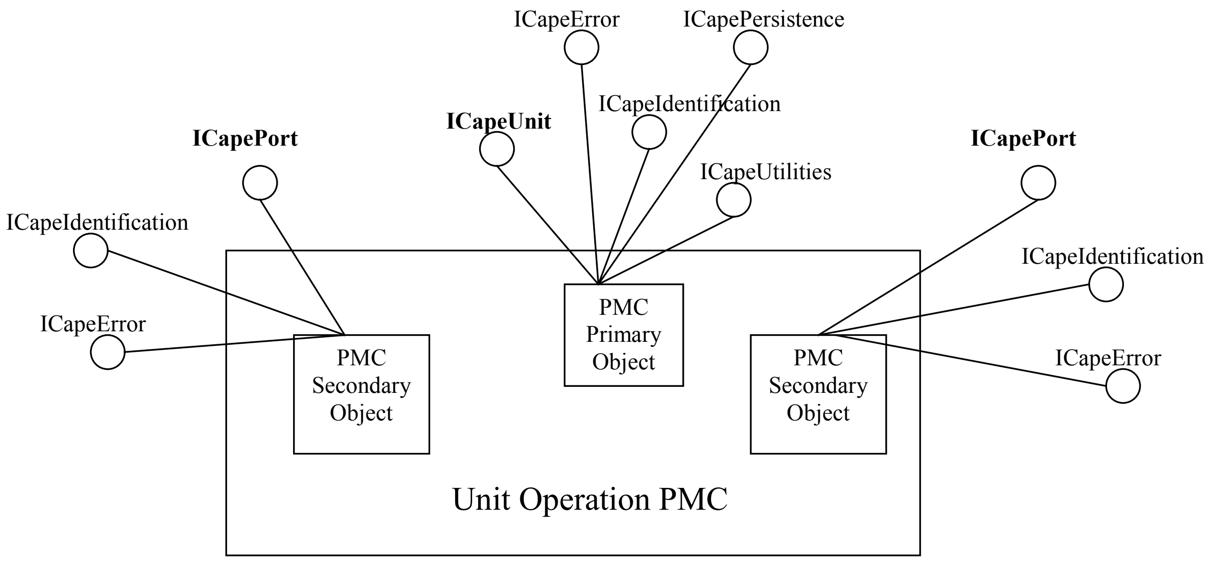


Figure 4 Component diagram for unit operation

Basically the Utilities functionalities are gathered within one single interface called ICapeUtilities. This interface is described in the next section.

Each business interface qualified as primary has to inherit from the ICapeUtilities interface. For example ICapeUnit which is implemented by a PMC primary object inherits from ICapeUtilities.

3.6 Interfaces descriptions

The ICapeUtilities interface is presented with its corresponding methods.

3.6.1 ICapeUtilities

Interface Name	ICapeUtilities
Method Name	GetParameters
Returns	CapeInterface (ICapeCollection)

Description

Returns an ICapeCollection interface. This interface will contain a collection of ICapeParameter interfaces. This method allows any client to access all the CO Parameters exposed by a PMC. Initially, this method was only present in the ICapeUnit interface. Since ICapeUtilities.GetParameters is now available for any kind of PMC, ICapeUnit.GetParameters is deprecated. Consult the “Open Interface Specification: Parameter Common Interface” document for more information about parameter. Consult the “Open Interface Specification: Collection Common Interface” document for more information about collection.

If the PMC does not support exposing its parameters, it should raise the ECapeNoImpl error, instead of returning a NULL reference or an empty Collection. But if the PMC supports parameters but has for this call no parameters, it should return a valid ICapeCollection reference exposing zero parameters.

Arguments

None

Errors

ECapeUnknown, ECapeFailedInitialisation, ECapeNoImpl

Interface Name	ICapeUtilities
Method Name	SetSimulationContext
Returns	--

Description

Allows the PME to convey the PMC a reference to the former's simulation context. The simulation context will be PME objects which will expose a given set of CO interfaces. Each of these interfaces will allow the PMC to call back the PME in order to benefit from its exposed services (such as creation of material templates, diagnostics or measurement unit conversion). If the PMC does not support accessing the simulation context, it is recommended to raise the ECapeNoImpl error.

Initially, this method was only present in the ICapeUnit interface. Since ICapeUtilities.SetSimulationContext is now available for any kind of PMC, ICapeUnit. SetSimulationContext is deprecated.

Arguments

Name	Type	Description
[in] simContext	CapeInterface	The reference to the PME's simulation context class. For the PMC to use this class, this reference will have to be converted to each of the defined CO Simulation Context interfaces.

Errors

ECapeUnknown, ECapeFailedInitialisation, EcapeInvalidArgument, ECapeNoImpl

Interface Name	ICapeUtilities
Method Name	Initialize
Returns	--

Description

Initially, this method was only present in the ICapeUnit interface. Since ICapeUtilities.Initialize is now available for any kind of PMC, ICapeUnit. Initialize is deprecated.

The PME will order the PMC to get initialized through this method. Any initialisation that could fail must be placed here. Initialize is guaranteed to be the first method called by the client (except low level methods such as class constructors or initialization persistence methods). Initialize has to be called once when the PMC is instantiated in a particular flowsheet.

When the initialization fails, before signalling an error, the PMC must free all the resources that were allocated before the failure occurred. When the PME receives this error, it may not use the PMC anymore. The method terminate of the current interface must not either be called. Hence, the PME may only release the PMC through the middleware native mechanisms.

Arguments

None

Errors

EcapeUnknown, ECapeOutOfResources, ECapeLicenceError, ECapeFailedInitialisation, ECapeBadInvOrder

Interface Name	ICapeUtilities
Method Name	Terminate
Returns	--

Description

Initially, this method was only present in the ICapeUnit interface. Since ICapeUtilities.Terminate is now available for any kind of PMC, ICapeUnit.Terminate is deprecated.

The PME will order the PMC to get destroyed through this method. Any uninitialization that could fail must be placed here. 'Terminate' is guaranteed to be the last method called by the client (except low level methods such as class destructors). 'Terminate' may be called at any time, but may be only called once.

When this method returns an error, the PME should report the user. However, after that the PME is not allowed to use the PMC anymore.

The Unit specification stated that "Terminate may check if the data has been saved and return an error if not." It is suggested not to follow this recommendation, since it's the PME responsibility to save the state of the PMC before terminating it. In the case that a user wants to close a simulation case without saving it, it's better to leave the PME to handle the situation instead of each PMC providing a different implementation.

Arguments

None

Errors

EcapeUnknown, ECapeOutOfResources, ECapeBadInvOrder

Interface Name	ICapeUtilities
Method Name	Edit
Returns	--

Description

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

Arguments

None

Errors

ECapeUnknown

3.7 Scenarios

4. Interface Specifications

4.1 COM IDL

```
// You can get these intructions in Common.idl file from CAPE-OPENv1-0-0.zip
```

4.2 CORBA IDL

```
// You can get these intructions in CAPE-OPENv1-0-0.idl within the
CAPEOPEN100::Common::Utilities module
```

5. Notes on the interface specifications

5.1 Parameters property

Indeed, the parameters interfaces have already been specified in a separate document. The issue here is to define which is the best way for a PME to access the PMC's parameters.

It would be useful if any kind of PMC could expose its parameters. The discarded alternatives would be:

- The parameters property is repeated in each interface that requires it. It prevents orthogonal use of CAPE-OPEN components.
- The components could implement the ICapeCollection interface. The design would not clearly express the contents of the collection.

A possible generalisation of the parameters property would be having a GetCollection(collectionName). in this way, GetCollection("parameters") would return the collection of parameters, and GetCollection("ports") would return the collection of ports.

5.2 CORBA IDL

From the conceptual model which is explained in the section 3, the implementation specifications section 4 for the CORBA platform is generated. There are some remarks to do:

- The edit operation is not currently supported. Indeed this approach that requires a same host and a MS-Windows OS is not compliant with the CORBA model.
- Each CORBA business interface qualified as primary has to inherit from the ICapeUtilities interface. Then the resulting PMC primary object will expose the Utilities functionalities. The ICapeUtilities interface acts as a base interface.
- The PME sets the reference of ICapeSimulationContextManager in the SetSimulationContext operation. The ICapeSimulationContextManager interface is defined within the Simulation Context COSE Interface specification document.

6. Prototypes implementation

7. Specific Glossary Terms

8. Bibliography

8.1 Process simulation references

(i) Open Interface Specification: Parameter Common Interface

(ii) Open Interface Specification: Simulation Context COSE Interface

(iii) Methods and Tools Integrated Guidelines

8.2 Computing references

8.3 General references

9. Appendices