CAPE-OPEN

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

.NET Interoperability Guidelines

www.colan.org

ARCHIVAL INFORMATION

Filename	Interoperability.doc
Authors	CO-LaN consortium
Status	Internal
Date	June 2006
Version	version 0.70
Number of pages	43
Versioning	Version 0.4 edited by Bill Barrett (US EPA)
	Version 0.5 edited by Michel Pons (CO-LaN)
	Version 0.6 edited by Lars von Wedel (AixCAPE)
	Version 0.61 edited by Lars von Wedel (AixCAPE)
	Version 0.62 proofread by Michel Pons (CO-LaN)
	Version 0.70 edited by Lars von Wedel and Bill Barrett
Additional material	
Web location	
Implementation specifications version	
Comments	

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.

Copyright © 2006 CO-LaN and/or suppliers. All rights are reserved unless specifically stated otherwise.

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

The CAPE-OPEN middleware standards were created to allow process modelling components (PMCs) developed by third parties to be used in any process modelling environment (PME) utilizing these standards. The CAPE-OPEN middleware specifications were based upon both Microsoft's Component Object Model (COM) and the Object Management Group's (OMG) Common Object Broker Architecture (CORBA). Since the inception of the CAPE-OPEN project, Microsoft updated COM to the .NET Framework. This document presents interoperability guidance and examples for .NET and provides a roadmap for the evolution of CAPE-OPEN into the .NET environment.

ACKNOWLEDGEMENTS

The CO-LaN consortium would like to thank the main authors, Bill Barrett and Lars von Wedel for the main work on the document. Further, we acknowledge Ms. Svetlana Strunjas for her help in preparing the figures included in the document. In addition, Bertrand Braunschweig and Jean-Pierre Belaud contributed to the document. Michel Pons supported the preparation of the standard layout. Finally, time and effort invested by the Method and Tools SIG members for reviewing, commenting, and improving the document are appreciated. We apologize for all remaining errors and ask for feedback to continually improve this document.

CONTENTS

ACRONYMS **9**

1. INTRODUCTION **10**

2. BACKGROUND **11**

3. A GENTLE INTRODUCTION TO THE .NET FRAMEWORK **12**

3.1 MANAGED CODE AND VIRTUAL MACHINES: THE ARCHITECTURE OF THE .NET FRAMEWORK 13
3.2 ASSEMBLIES – UNITS OF CODE CONTAINING METADATA 13
3.3 THE .NET WAY OF LANGUAGE INTEROPERABILITY 15
3.4 INTEROPERABILITY WITH UNMANAGED CODE 17
3.4.1 Platform Invocation Service *18*
3.4.2 Interop Assemblies – Enablers for COM Integration *18*
3.4.3 Using COM components in .NET *19*
3.5 IMPLEMENTING COM COMPONENTS IN .NET 21

4. CAPE-OPEN SPECIFIC INTEROPERABILITY **23**

4.1 DATA TYPES AND VALUES 23
4.1.1 A Comparison of COM and .NET Data Types *23*
4.1.2 Defining Interfaces *25*
4.1.3 Importing and Exporting Type Libraries *27*
4.2 ERROR HANDLING 28
4.2.1 Exception Handling for a COM-Based PMC in a .NET-based PME *28*
4.2.2 Throwing Exceptions from a .NET-Based PMC in a COM-based PME *29*
4.3 IMPLEMENTATION OF COLLECTIONS 31
4.4 PERSISTENCE 32
4.4.1 Persistence of a .NET-based PMC in a COM-based PME *32*
4.4.2 Persistence of a .NET-based PMC in a .NET-based PME *34*
4.4.3 Persistence of a COM-based PMC in a .NET-based PME *34*
4.5 REGISTERING OBJECTS 34

5. MOTIVATION AND ROADMAP FOR .NET-BASED CAPE-OPEN **38**

6. BIBLIOGRAPHY **40**

7. APPENDIX CREATING AND THROWING .NET EXCEPTIONS **41**

LIST OF FIGURES

FIGURE 1 .NET FRAMEWORK ARCHITECTURE 12
FIGURE 2 ASSEMBLY CONTENTS 14
FIGURE 3 HIERARCHY OF CORE TYPES (TROELSEN 2002) 16
FIGURE 4 .NET LANGUAGE INTEROPERABILITY 17
FIGURE 5 PLATFORM INVOKE CALL TO UNMANAGED DLL 18
FIGURE 6 .NET RUNTIME CALLABLE WRAPPERS (BUSBY AND JEZIERKSI 2001) 20
FIGURE 7 .COM CALLABLE WRAPPERS (BUSBY AND JEZIERKSI 2001) 22

LIST OF TABLES

TABLE 1. COMPARISON OF CAPE-OPEN, .NET AND COM DATA TYPES. 23
TABLE 2. COMMON HRESULT ERROR VALUES AND CORRESPONDING .NET EXCEPTIONS 28

Acronyms

.exe .... Executable program
.NET .... Microsoft's .NET Framework
API .... Application Programming Interface
ASP............................... Active Server Pages
BSTR .... Basic String
C# .... C# (C-Sharp) Programming Language
C++ .... C++ Programming Language
C++/CLI .... Common Language Infrastructure extensions to the C++ Programming Language
CAPE............................ Computer Aided Process Engineering
CCW .... COM-callable wrappers
CIL................................ Common Intermediate Language
CLI................................ Common Language Infrastructure
CLR .... Common Language Runtime
CLSID .... Class Identification GUID
CO-LaN .... CAPE-OPEN Laboratories Network (http://www.colan.org)
COM .... Microsoft's Component Object Model
CORBA .... Common Object Request Broker Architecture
CPU .... Central Processing Unit
CTS............................... Common Type System
DLL or .dll.................... Dynamic Linked Library
ECMA .... European Computer Manufacturers Association
FAT .... File Allocation Table
GAC.............................. Global Assembly Cache
GUID .... Globally Unique Identifier
HRESULT .... COM Error Result
idl.................................. Interface Definition Language
IJW .... "It Just Works"
IL .... Intermediate Langange
ISO................................ International Organization for Standardization
Java .... Sun Microsystems's Java Programming Language
JIT................................. Just-In-Time
Mono .... Mono Project (http://www.java.com)
MSIL .... Microsoft intermediate language
NTFS .... NT File System
OMG............................. Object Management Group (http://www.omg.org)
P/Invoke or P/I.............. Platform Invoke Services
PDA .... Personal Digital Assistant
PIA................................ Primary Interop Assembly
PMC.............................. Process Modelling Components
PME.............................. Process Modelling Environment
RCW .... Runtime-callable Wrappers
SAFEARRAY .... COM Array Type.
SOAP............................ Simple Object Access Protocol
STL............................... Standard Template Library
UDDI .... Universal Description, Discovery and Integration
VB .... Visual Basic Programming Language
WSDL........................... Web Services Definition Language
XML .... Extensible Mark-up Language

1. Introduction

This document addresses technical aspects of interoperating .NET and COM-based software components and environments through CAPE-OPEN interfaces. It is mainly targeted at process modeling components (PMC) and process modeling environments (PME) designers and developers. The "Gentle Introduction to the .Net Framework" section can be read by anyone interested in the general principles.

The document assumes some general knowledge of the CAPE-OPEN standard and of Microsoft COM programming, including familiarity with principles of object-oriented software and with at least one of the programming languages used as examples (C# or C++/CLI). Developers unfamiliar with CAPE-OPEN can start by looking at the CAPE-OPEN standard (http://www.colan.org), CAPE-OPEN’s Methods and Tools Integrated Guidelines (http://www.colan.org/Spec 10/Methods&Tools Integrated Guidelines.pdf) and specific CAPE-OPEN interface specifications (http://www.colan.org/index-15.html) such as Unit Operations, Physical Properties, and Common Interfaces.

Code examples provided in this document are mostly part of existing and working .NET PMC or PME implementations. Thus, summarizing the remainder of the document, we can act on the assumption that interoperability between code written in .NET and existing implementations of PMEs and PMCs following the CAPE-OPEN standard defined in COM interfaces can be achieved. Due to a lack of resources not all interfaces existing in the CAPE-OPEN standards body have been evaluated or even tested in implementations. However, the *common interface specifications* as well as important *business interface specifications* have been successfully implemented in an interoperability context. Hence, the authors strongly believe that interoperability .NET is a workable solution for the period of time in which COM and .NET implementations will exist besides each other. Further, the authors are aware that performance may be an issue in certain process simulation applications. Future case studies shall therefore reveal performance behavior of interoperability solutions with COM- and .NET-based components.

The document is organized as follows: Sections 2, 3 and 5 give clues on the feasibility for CAPE-OPEN to move to .NET; section 4 shows how to do it for knowledgeable developers. More specifically Section 2, Motivations, explains why interoperability between .NET and COM is needed. Section 3 "a Gentle ..." explains the principles of .NET in a few pages and can be read by everyone. Section 4 goes into the technical details of COM/.NET CAPE-OPEN interoperability in different flavors, using the existing CAPE-OPEN interface specifications. Section 5 examines options for developing native .NET specifications for CAPE-OPEN and making them interoperable with the existing .COM specifications. They are followed by a list of references.

2. Background

The CAPE-OPEN program was begun in 1995 to develop, test, describe and publish agreed standards for interfaces of components of a process simulator. Its objectives are to enable native components of a simulator to be replaced by those from another independent source or a part of another simulator with a minimum of effort in as seamless a manner as possible (CAPE-OPEN 2000). This document is intended to provide guidelines for implementing the agreed standard interfaces in the .NET framework not provide the description and design details of the specific interfaces. For more information on the design of the specific interfaces, see the CO-LaN Website (www.colan.org). Background of the CAPE-OPEN projects can be found at (Pons 2003).

At that time, two competing object models were available, Microsoft's Component Object Model (COM) (Microsoft 1995) and the Object Managements Group's Common Object Request Broker Architecture (CORBA) (see http://www.omg.org). The advantage of using these object models in developing applications is that the application can function as platform-independent, object-oriented system for creating binary software components that can interact with other components in the same process space or in other processes on remote machines. These object-oriented programming models not only allowed the insertion of third-party objects into an application, but these models have been extended to enable objects to be accessed over a network or even the internet.

Concurrent with the development of the CAPE-OPEN standards, there has been significant growth in the use of the internet and distributed computing, and both COM and CORBA have evolved to meet the requirements of distributed computing environments. Microsoft has updated COM to a framework called the .NET Framework. In doing so, Microsoft has included a number of technologies for creating and working in a distributed environment such as remote object access; extensible markup language (XML) tools; the Simple Object Access Protocol (SOAP); universal description, discovery and integration (UDDI); and web services, which use the web services definition language (WSDL), in the .NET Framework.

The .NET Framework is essentially an update of COM, and it readily interoperates with COM objects with little problem. Indeed, the C++ compiler automatically adds the code needed for COM/.NET interoperation through the "It Just Works" or IJW technology (Grimes 2003). The Microsoft Developers Network (MSDN) provides detailed information regarding COM/.NET interoperation at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconinteroperatingwithunmanagedcode.asp

This document will discuss interoperability issues with respect to COM and .NET-based Process Modelling Environments (PME) and Process Modelling Components (PMC). The discussion will consider both the insertion of a .NET-based PMC into a legacy COM-based PME, the immediate issue, and the longer term issues of using legacy COM-based PMCs in a .NET-based PME.

3. A Gentle Introduction to the .NET Framework

The Microsoft .NET framework was created around the late 1990s by Microsoft with several goals in mind, which included the following:

- Unification of the various development technologies being used to date (such as COM, Active Server pages (ASP), etc.)
- Bringing an opponent to Sun's Java technology on the market
- Better coverage of mobile devices
- Simplified deployment of applications (fighting so called *DLL hell*)
- Better response to security issues

The major difference as opposed to current software development technologies is the introduction of so called managed code which is not executed by a physical processor in hardware, but by a virtual processor emulated by a piece of software called a virtual machine. This will be explained in more detail in Section 3.1. The code to be executed by virtual machines resides in so-called assemblies (cf. Section 3.2) which resemble dynamic linked libraries (DLLs) but are in addition equipped with metadata describing their identity, locale, version number, content, and many other things. The virtual machine of the Common Language Runtime provides a type system which permits data and classes to be shared across software written in a variety of several programming languages (cf. Section 3.3). Finally, this chapter will describe means of interoperability that Microsoft has made available to interoperate with legacy DLLs and COM components (cf. Section 3.4).

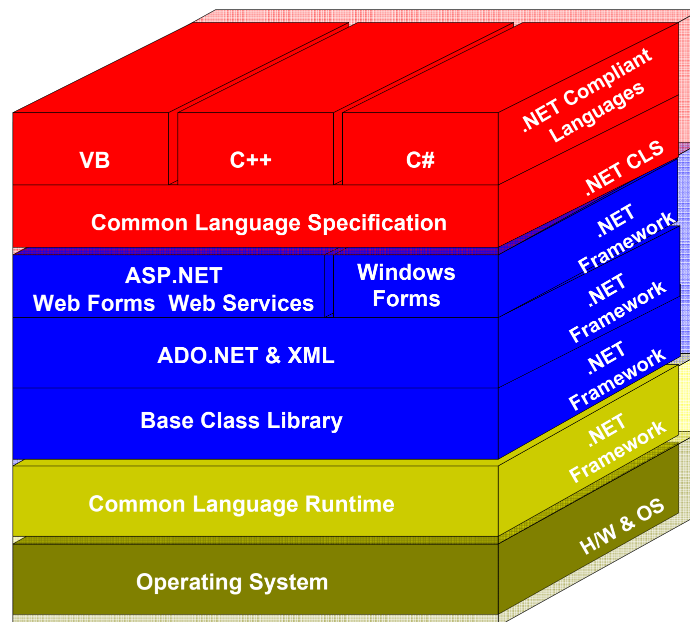

Figure 1 .NET Framework Architecture

3.1 Managed code and virtual machines: the Architecture of the .NET Framework

The architecture of the .NET framework is based on an open specification of the Common Language Infrastructure (CLI) that was ratified as an open standard with the European Computer Manufacturer's Association (ECMA) (ECMA 2005) and has been submitted to the International Organization for Standardization (ISO) . This standard specification has not only been used as a basis for implementing the Microsoft .NET framework, also other projects such as the Mono Project (http://www.mono-project.com/Main_Page) development platform (for various Unix operating systems variants) or Portable .NET (http://www.dotgnu.org/pnet.html) have been developed according to this specification.

At the heart of the .NET framework, the common language runtime (CLR) implements a virtual machine that interprets byte code, a hardware independent intermediate format between high-level languages such as Basic or C++ and the machine code that is executed by the central processing unit (CPU) in hardware. The major advantage of such an approach is that changes in the underlying hardware can be accounted for by developing or adapting a single virtual machine instead of modifying numerous existing software packages. Use cases that come to mind is the upcoming shift towards processors using 64 bits for data and address representation, dual core processors or mobile devices such as personal digital assistants (PDAs) or smartphones. When an application is used on such platforms, the bytecode will remain the same, only the processing of the bytecode on the target platform differs in that it takes into account the peculiar properties of the platform.

The byte code that results from the compilation of an application is not interpreted directly, but run through a Just-In-Time (JIT) compiler that executes when an application is started. The speedup of executing compiled code vs. interpreting byte code by far outweighs the overhead of the just-in-time compilation step itself. In summary, the performance of .NET-based applications is about the same as for applications developed using unmanaged code.

Another advantage of executing an application in a virtual environment instead of granting access to the physical hardware directly is the fact that the executing CLR can guard the memory usage of the application and perform a so-called garbage collection that frees objects no longer referenced by the application. The result is an improved stability and safety of the application as well as a reduced effort on behalf of the developer to spend time in tedious reference counting, for example.

The byte code for the .NET framework is specified in a language called common intermediate language (CIL), sometimes also termed Microsoft intermediate language (MSIL). Type definitions (such as data and classes) in the intermediate language rely on the common type system (CTS) discussed further in Section 3.3.

3.2 Assemblies – Units of Code containing Metadata

Compiled code in managed form is contained within an assembly, a file with either.exe (executables) or .dll (dynamic link libraries) extension. However, the contents have completely changed: the contained code is byte code (see above) as opposed to machine code and metadata, a set of information that describes the contents of the assembly.

Assemblies usually consist of four elements: the assembly metadata (also called a manifest), metadata describing the types, the Intermediate Language (IL) code that implements the types and a set of resources. All elements except the manifest are optional. These elements can be combined in different ways, depending on whether a single .dll file contains an entire assembly, or the assembly content is spread across multiple files (see Figure 2).

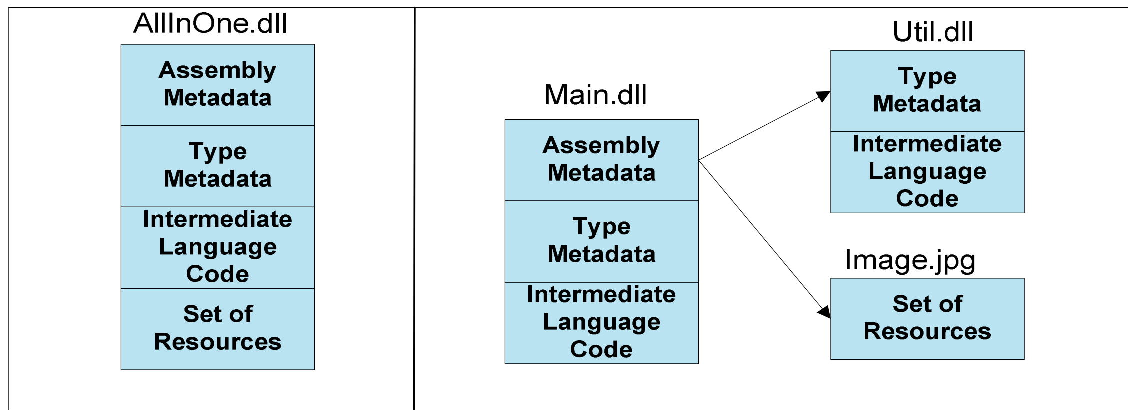

Figure 2 Assembly Contents

An important item of a manifest is the assembly identity which is more than just a filename. The assembly identity comprises a short name (which is used for the file name), an optional culture (to permit assemblies to come in localized versions), a version identifier, and an optional public key token which is a hash value paired with a private key (it is used if the assembly is built for sharing). An assembly having a public key token is said to be strongly named (or signed) and cannot be exchanged against a malicious assembly (spoofing) without having access to the matching private key. In addition to the assembly identity, every manifest includes the following data:

- **File List:** The file list includes a list of all files that make up the assembly. For each file, the manifest records its name and cryptographic hash of its content, which is checked and verified at run time to ensure that deployment unit is consistent.

- **Referenced assemblies:** Referenced assemblies store information about dependencies between assemblies. These dependency pieces of information include a version number, that is used at run time to ensure the correct version of dependency is loaded.

- **Exported Types:** Each assembly contains metadata about all of contained types and resources within itself so that application developers can inspect the contents of the assembly by a technology called reflection. An example use is to enumerate all types contained in an assembly to check whether they implement a certain interface, such as *ICapeUnit*. This approach can effectively be used to circumvent registration of assemblies in a central place such as the registry. Instead, assemblies can be placed in a shared directory where applications looking for certain components can iterate assemblies present and easily find out which assemblies contain types that are useful for the application.

- **Permission Requests:** There are three type of permission requests: those required for the assembly to run, those that are desired but the assembly will still have some functionality without them and those that the author never wants to be granted. In addition to the reflection capabilities, metadata can contain *custom attributes* which are used to flag a type with certain properties. There is a predefined set of so-called *pseudo custom attributes* which is known to the CLR and controls the way the CLR works with these types.

Examples (relevant to this document) are attributes that control the interaction of types with COM-based applications:

```
[
	ComVisible(true),
	ProgId("DistillationShortcutUnitAixCAPE")
]
public class DistillationShortcutUnit
{
	// …
}
```

The attribute `ComVisible` instructs the CLR to make this class available via COM (cf. 3.4 Section for more details) and the `ProgId` attribute sets the ProgId of the class.

Assemblies can generally reside anywhere on the filesystem, however, the location must be known for an application to be able to access them. Strongly-named assemblies can be placed in a central area called global assembly cache (GAC) which is specifically designed to hold assemblies that are to be shared by different applications. The purpose of the GAC is to avoid having different copies of a single assembly at various location on the machine and to permit various versions of an assembly with the same name to coexist besides each other (this is not possible within one directory of a FAT or NTFS filesystem). Assemblies can be added to the GAC using the GAC utilities tool (gacutil.exe) contained in the .NET framework.

3.3 The .NET Way of Language Interoperability

When using COM as a means to make applications developed using different programming languages interoperable the developer had to take care of mapping native datatypes into COM datatypes and back. As an example C++ developers had to convert character pointers (char*) or STL strings (std::string) into a string type defined by COM that was called BSTR. Sequences in C++ (e.g. as a simple array or a list type) had to be mapped back and forth between a SAFEARRAY.

The .NET framework instead takes a different approach to solve this problem. The common language runtime provides a set of types common to all programming languages that are based on .NET – this is called the *common type system* (CTS). By sharing these type definitions from ground up, language interoperability becomes a rather simple issue because no conversions are necessary any longer.

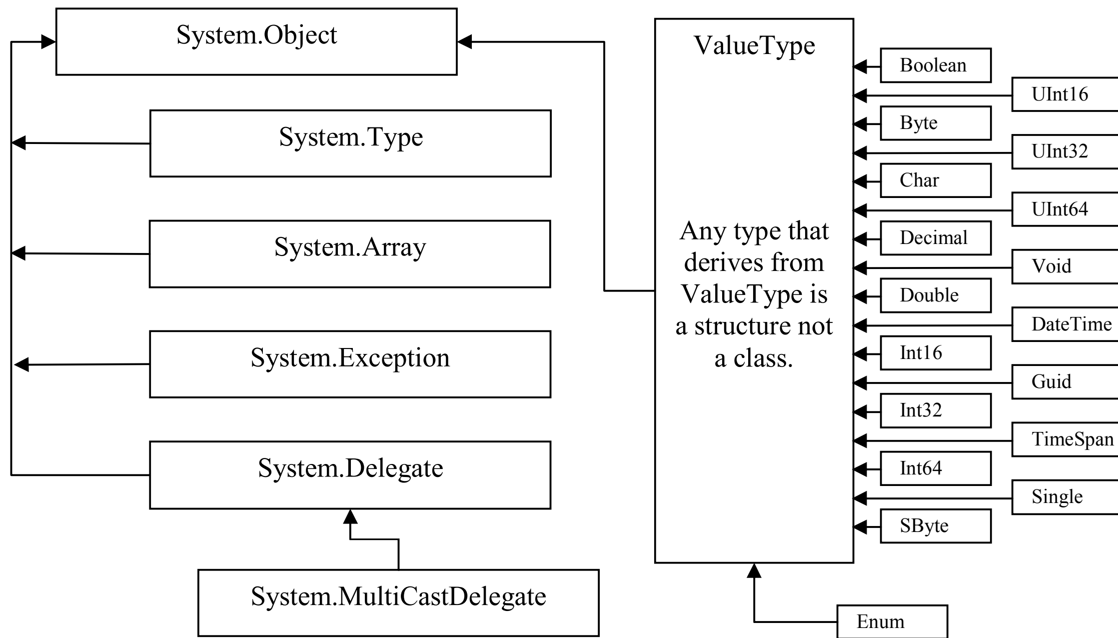

Figure 3 Hierarchy of Core Types (Troelsen 2002)

In addition, all languages share a representation of their code in the intermediate language (CIL/MSIL) which basically means that all function or method definitions are present in a compatible form as well. Hence, calling functions and methods across language or application boundaries is also a rather simple issue and is completely taken care of by the common language runtime. In a sense, the same algorithm written in different programming languages, such as C#, Visual Basic (VB), or C++, can be regarded as different views on the same problem because the underlying intermediate language is basically identical.

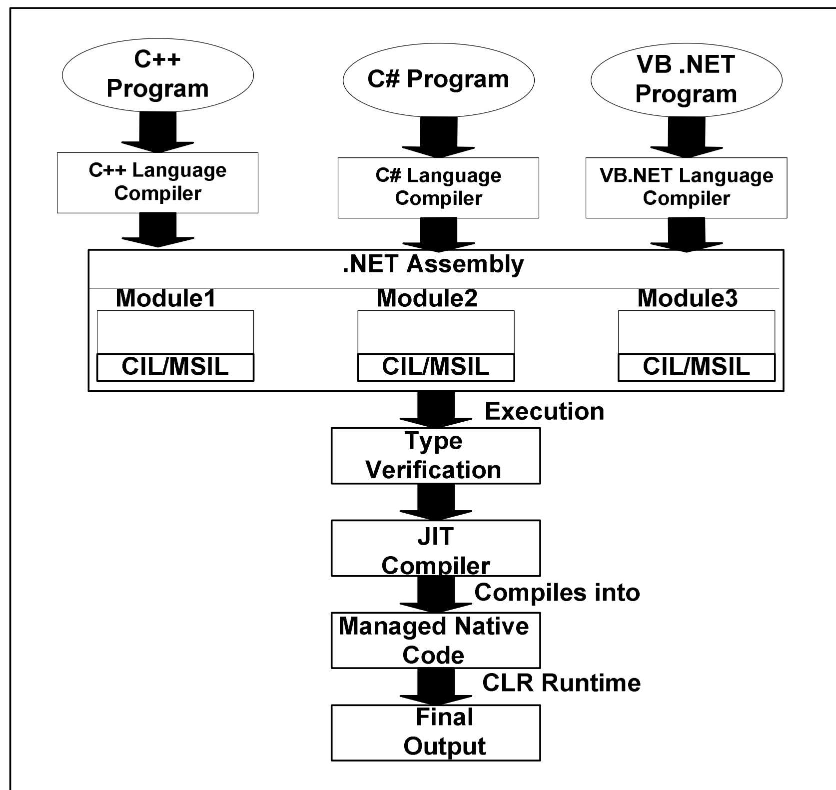

Figure 4 .NET Language Interoperability

As a consequence, the language in which components have been constructed becomes completely transparent to developers. They can not only call methods on behalf of a foreign component regardless of the languages involved in the implementation, but they can even inherit from code written in other languages (including method overloading) and issues such as serialization or exception handling (see Section 4.2 for more detail) are handled by the CLR in a completely transparent fashion.

3.4 Interoperability with Unmanaged Code

Obviously, a lot of code was written before the .NET framework came into place and it is unrealistic to believe that all of this code can or will be migrated in a short period of time. Hence, Microsoft has put substantial effort into mechanisms that permit interoperability of code provided for the .NET environment (managed code) with so-called *unmanaged code* – classic DLLs that were developed using C++ or Fortran in times before .NET or COM components that have been developed to permit interoperability of applications.

Basically, there are four mechanisms to be mentioned, three of which will be explained in more detail in the subsequent sections:

- *Unsafe code* permits the immediate use of pointers into the data address space of the application. It is used in rather special cases and only listed here for the sake of completeness.
- *Platform invoke* services permit to call functions/procedures in unmanaged DLLs (see Section 3.4.1).
- *COM-callable wrappers* (CCWs) make a .NET component available to COM-based applications (see Section 3.4.2).
- *Runtime-callable wrappers* (RCWs) allow a COM-based component to be called by a .NET application (see Section 3.4.3).

3.4.1 Platform Invocation Service

Calling methods in an existing DLL made from C or FORTRAN is a simple task using the so-called *platform invocation services*, also called *P/Invoke* for short. Since these DLLs do not have any metadata associated with them, the developer has to supply a representation of the methods in a .NET-based programming language. An example demonstrates how a routine *calc_flash* in the library thermo.dll can be accessed from a .NET-based language (here C#):

```
class ThermoWrapper
{
	[DllImport("thermo.dll")]
	internal static extern int
calc_flash(double p, [In, Out] double[] z,
double v, IntPtr user_data);
}
```

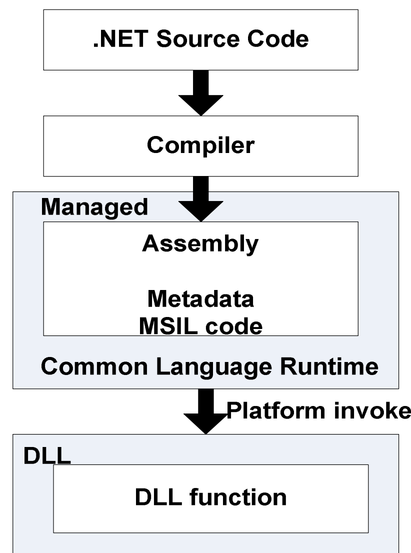

Figure 5 Platform Invoke Call to Unmanaged DLL

This example introduces a class *ThermoWrapper* which is basically just a dummy in order to hold the method *calc_flash* because C# does not permit methods to be defined outside the context of a class. The custom attribute *DllImport* (cf. Section 3.2) instructs the common language runtime to look inside the referenced DLL for the implementation of the method. The method is declared as internal (visible only inside the assembly), static (not bound to an object instance), extern (not implemented in this assembly) and returns an integer after terminating. The parameters the method takes are a scalar double, a double array (which matches e.g. to a double array in C) and is input and output to/from the method, a second scalar double, and finally a pointer to some arbitrary user data storage (which matches e.g. a void pointer in C). Attributes for parameters (such as the *[In,Out]*) exert fine-granular control over the process of transforming the data back and forth between the managed and the unmanaged execution. Similar mappings permit the interoperability with a wide range of data types and languages.

3.4.2 Interop Assemblies – Enablers for COM Integration

Integrating .NET with COM is even simpler than the above mentioned integration through P/Invoke because a type library for a COM object holds metadata that can be used to actually generate a suitable wrapper instead of having to write stub methods as in the previous example.

Basically, there are two general approaches how to interoperate with COM applications. An existing COM component can be used as an add-in within a .NET application or .NET can be used to develop a COM-based plug-in. Both approaches are supported equally well by the .NET framework. In both cases, an intermediate piece of software to translate between .NET and COM is needed and is called an *interop assembly*. A particular type of interop assembly is the *primary interop assembly* (PIA) which is signed by the component publisher and marked as belonging to the COM component. A PIA can be registered with the GAC and will be used by default whenever a .NET application is going to use the COM component to which the PIA belongs [see (Nathan 2002) or (Troelsen 2002) for details].

3.4.3 Using COM components in .NET

When a COM component is to be used in a .NET application an intermediate piece of software is needed that translates between the .NET set of types (according to the CTS, cf. Section 3.3) and the types that are defined in COM (SAFEARRAY, BSTR, ...) and exposes the interfaces implemented by the COM components in a .NET-compatible way. This piece of software is termed a *runtime callable wrapper* (RCW for short) because the common language runtime can natively call the wrapper around the COM component.

Luckily, this intermediate piece of software can be generated in a fully automated fashion from a type library which is usually present for COM components that are meant to be used by other applications. The .NET framework contains the tool *tlbimp.exe* which is given a type library and generates an assembly containing the RCW from it.

This assembly becomes part of the application using the COM component. It contains all types derived from the type library inside a namespace which is equivalent to the type library name, e.g. *CAPEOPEN100*. In addition, classes are generated for *CoClass* entries in the type library so that instantiating and using a COM component becomes a simple activity. The following C# code snippet exemplifies how to launch an instance of Microsoft Excel (through the PIAs delivered by Microsoft), load a workbook from a .xls file and activate it:

```
Microsoft.Office.Interop.Excel.Application excel = null;
Microsoft.Office.Interop.Excel.Workbook wb = null;
object  missing  = Type.Missing;

excel = new Microsoft.Office.Interop.Excel.Application();
wb = excel.Workbooks.Open("c:\\test.xls", missing, missing, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing, missing, missing);
excel.Visible = true;
wb.Activate();
```

The CAPE-OPEN type library however does not contain any CoClass definition (remember that CAPE-OPEN is only about interface definitions). However, a COM-based property system for example implements CAPE-OPEN interfaces and can easily be used with the imported interop assembly:

```
Type type = Type.GetTypeFromCLSID( new System.Guid( "<CLSID>" ) );
object ts_obj = Activator.CreateInstance( type );

ICapeThermoSystem ts = ts_obj as ICapeThermoSystem;
string[] packages = (string[])ts.GetPropertyPackages();

object pp_obj = ( ts.ResolvePropertyPackage( packages[0] );
string name = (pp_obj as ICapeIdentification).ComponentName;
string[] compids = (string[]) package.GetPropList();
```

The example (C#) demonstrates how to instantiate a property system through the registry from a given class identifier (CLSID). Then, the obtained object is cast into a thermo system reference and the identifiers of available property packages are obtained through a call to GetPropertyPackages. Finally, a property package is resolved (call to *ResolvePropertyPackage*) and the component name is obtained via the *ComponentName* property of the *ICapeIdentification* interface. In addition, a list of available properties (*GetPropList*) is

obtained. The same output can be achieved by any .NET-compliant language such as Visual Basic, C++ and even less common languages.

Actually, when a COM component is used in the .NET environment the RCW consumes the interfaces offered by the COM component and presents them in a way convenient for use within .NET. Custom interfaces (such as *ICapeUnit*) are translated according to a set of well-defined rules (Troelsen 2002), which will be further discussed below. Additionally, a number of well-known interfaces are handled in particular ways. Important examples in this category are:

- *IUnknown*: The RCW makes use of the *IUnknown* interface to support type casting and lifecycle management. As a result, calls to *QueryInterface* or *AddRef/Release* are never necessary from within .NET code.

- *IDispatch*: Late binding to COM objects is made available through the reflection services in the .NET world. Thus, capabilities offered through the *IDispatch* interface are accessible in a transparent fashion in .NET.

- *IConnectionPointContainer*: Connection points in COM that are registered through the interface *IConnectionPointContainer* are made available as native events in .NET through so-called delegates (essentially callback routines).

- *IErrorInfo*: The COM way of transmitting error information to a client is emulated and errors are mapped to structured exceptions in any of the available .NET languages, as described below.

- *IENUMVariant*: Collections made available by the COM components are also wrapped in a way that permits their access through native .NET interfaces.

As a consequence, an object made available by a COM component looks very similar to a native .NET object and is therefore easily integrated.

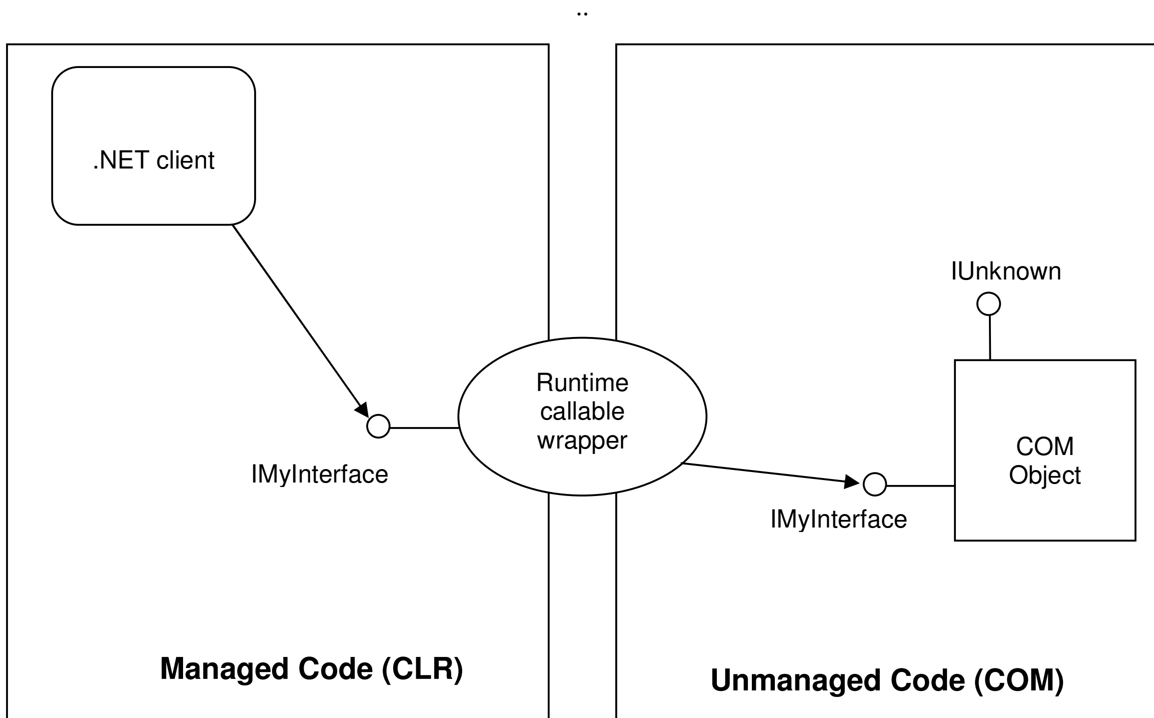

Figure 6 .NET Runtime Callable Wrappers (Busby and Jezierksi 2001)

3.5 Implementing COM Components in .NET

This section gives a brief overview about the implementation of COM components using the .NET framework which is kind of the inverse perspective described in the former section. Again, the .NET framework reduces the effort to be spent by the developer by automatically generating an intermediate piece of software that deals with the interaction of the .NET component code and the COM runtime system. This generated wrapper is called a *COM callable wrapper* (CCW) and implements COM interfaces based on the properties of the underlying code developed using .NET. The main contribution by the CCW is to translate types between the common type system and the COM type definitions, to handle reference counting, type casting, and to implement well-known interfaces already defined in COM such as *IDispatch*, *IUnknown*. In addition, the public methods of a class in .NET are exported as methods of an interface of the corresponding COM object. Collections will be made available through the *IEnumVARIANT* interface and delegates (event callbacks) will be represented through the interface *IConnectionPointContainer*.

In many cases, it is not enough to make a .NET class available as an arbitrary interface. Usually, some software has defined a COM interface that permits extensibility of an application such as Add-Ins for MS Excel or units and property systems for a CAPE-OPEN compliant simulator. In these cases it is important that the interfaces exposed by the CCW mirror exactly the COM interfaces expected by the application. There are basically two ways how this can be achieved:

- The .NET component code can reference the interop assembly imported from an existing type library (cf. former section) and implement the interfaces defined in the type library.
- The code can be written using .NET and the CCW generation is controlled by adding metada attributes so that the interfaces exposed by the CCW mirror exactly what is expected by the COM client application.

Let us briefly look into code example that explains how to implement the mandatory *ICapeIdentification* interface using the first approach, assuming that the type library importer tlbimp.exe has been used to import the CAPE-OPEN interface definitions into an interop assembly that provides the namespace CAPEOPEN100 with all CAPE-OPEN interface and type definitions:

```
[
	ComVisible(true),
	ProgId("COComponent"),
	ClassInterface(ClassInterfaceType.AutoDual)
]
public class COComponent : CAPEOPEN100.ICapeIdentification
{
	private string m_name;
	private string m_description;

	public string ComponentName
	{
		get	{	return m_name;		}
		set	{	m_name = value;		}
	}

	// same for component description
}
```

The example defines a class entitled ***COComponent*** that implements the interface *ICapeIdentification* defined in the *CAPEOPEN100* namespace. The class is equipped with three attributes:

- The attribute *ComVisible* controls whether the class is exported to the COM runtime system at all. Even if this attribute is set to true, only public classes are visible.
- The *ProgId* attribute sets the program id of the component.
- The *ClassInterface* attribute is used to determine whether a class interface is exported. The setting *AutoDual* permits early as well as late binding of COM clients to this object.

Within the class the string-valued property Component Name is defined. Properties are a language element in C# and are mapped immediately to properties of COM objects.

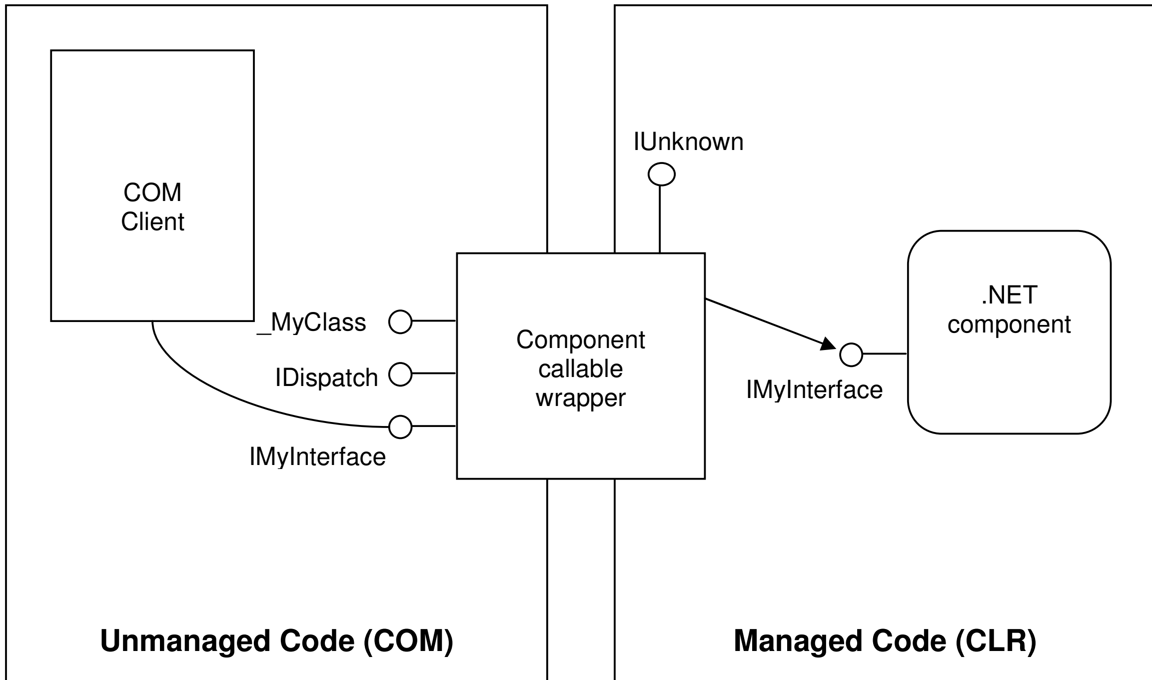

Figure 7 .COM Callable Wrappers (Busby and Jezierksi 2001)

4. CAPE-OPEN Specific Interoperability

Various implementation issues arise when a complex application programming interface (API) such as CAPE-OPEN is updated from one version of middleware to another. While COM interop discussed above generally works and handles most cases, as expected, there are some areas where .NET implementations must be aware of the peculiarities of COM to properly interoperate. Obvious issues are data types/values, exceptions and persistence where .NET has made significant departures from COM. While these issues are generally well handled in COM interop, awareness of some of the implementation details can be helpful. The following sections discuss some of these items.

4.1 Data Types and Values

As described above,.NET uses a common type system (CTS) where the type of an object is the primary issue. Classes, interfaces, structures, and data all have a particular type defined within the assembly. While this is similar to the COM type library and interface definition language (idl) code, some variations exist and the differences will be discussed below.

4.1.1 A Comparison of COM and .NET Data Types

CAPE-OPEN defines a number of data types within the idl files and these types are used to define the interfaces. Some of these types are simple C++ types such as long integer, float, or double, while others are COM-specific such as Visual Basic strings (BSTR) and Variants. Further, CAPE-OPEN defines a number of enumerations such as the *CapeValidationStatus* that need to be translated into a .NET data type.

COM interop basically translates the COM data types into their equivalent .NET data types. Table 1 provides a list of the CAPE-OPEN data types, their COM equivalents and the data type that the interop marshaller would provide to .NET.

Table 1. Comparison of CAPE-OPEN, .NET and COM Data Types.

CAPE-OPEN Data Type	Description of Data Type	COM Data Type	.NET Data Type
CapeLong	long	long	long
CapeShort	short	short	short
CapeDouble	double	double	double
CapeFloat	float	float	float
CapeBoolean	bool	VARIANT_BOOL	bool
CapeChar	char	char	char
CapeString	String or char[ ]	BSTR	System.String
CapeDate	date	VARIANT_DATE	System.DateTime
CapeURL	URL string	BSTR	System.String

CapeVariant	void	Variant	Object
CapeInterface	CO Interface	LPDISPATCH	Object
CapeArray(TYPE)	Array of (Type)	Variant containing SafeArray(Type)	(Type) [ ]

Of the data types listed, many are handled directly by COM interop. The *CapeBoolean* and *CapeInterface* data types require the *MarshalAs* attribute to be applied to the function header to be properly marshalled, as described below. *BSTR*s are automatically marshalled to/from *System.String* and the *CapeDate* data type is automatically marshalled to *System.DateTime*. Values and arrays are automatically boxed (converted to objects) and converted to the appropriate *Variant* types in COM interop.

While arrays are automatically boxed, COM-based CAPE-OPEN requires the array to be wrapped in a variant. This process results in loss of type information for the array on the .NET side of the interop. What this means is if an array of integers is desired, the .NET function header should use an *Object* data type, not an *int[]* data type. If the *int[]* data type is defined, the type library exporter will create idl using the *SAFEARRAY(int)* as the data type, not the CAPE-OPEN specified *Variant*. Below is a C++/CLI implementation of the *ICapeThermoMaterialObject::GetUniversalConstant*() method showing a *CapeArrayString* input parameter and the return of a *CapeArrayReal*:

```
// Get some universal constant(s)
//
// CAPE-OPEN exceptions
// ECapeUnknown, ECapeInvalidArgument, ECapeNoImpl
// [id(3), helpstring("method GetUniversalConstant")]
// HRESULT GetUniversalConstant([in] CapeArrayString props,
//                              [out, retval] CapeArrayVariant *propVals);
Object^ CapeOpen::CCapeThermoMaterialObject::GetUniversalConstant(Object^ props)
// input is array<String^>^, return value is array<Object^>^.
{
	array<String^>^ propList = dynamic_cast<array<String^>^>(props);
	if (!propList) return nullptr;
	int numProps = propList->Length;
	array<double>^ value = gcnew array<double>(numProps);
	for (int i = 0; i < numProps; i++){
		bool constantInList = false;
		if (!(String::Compare("avogadroConstant", propList[i]))){
			value[i] = double(6.022141995e23);
			constantInList = true;
		}
		if (!(String::Compare("boltzmannConstant", propList[i]))){
			value[i] = double(1.3806503e-23);
			constantInList = true;
		}
		if (!(String::Compare("molarGasConstant", propList[i]))){
			value[i] = double(8.314472);
			constantInList = true;
		}
		if (!(String::Compare("standardAccelerationOfGravity",
						propList[i]))){
			value[i] = double(9.80665);
			constantInList = true;
		}
		if (!(String::Compare("faradaysConstant", propList[i]))){
			value[i] = 9.64846e-19;//coulombs per mole
			constantInList = true;
		}
		if (!(String::Compare("chargeOfElectron", propList[i]))){
			value[i] = 1.60219e-19;//coulombs
			constantInList = true;
		}
		if (!constantInList) throw gcnew CapeInvalidArgumentException(
```

```
                      String::Concat(propList[i], " is not a Universal Constant in
                      CCapeThermoMaterialObject::GetUniversalConstant"), 1);
        }
        return value;
}
```

Empty variants are defined in .NET as *null*-valued objects (C# – *null,* C++/CLI – *nullptr*). CAPE-OPEN methods that require empty variants should have the value *null* placed in the function call. On the .NET side, an empty variant sent by a COM object will translate into a *null* valued object. In the above implementation, the second line of code "`if (!propList) return nullptr;`" tests for an empty variant and returns an empty variant if one is sent as an argument.

Enumerations such as the *CapeValidationStatus* also need to be defined for COM interop. The .NET framework defines enumerations as a *enum* class. As a result, the *CapeValidationStatus* enumeration is defined as follows in .NET:

```
// .NET Translation of Validation Status Values.
[
        ComVisibleAttribute(true),
        GuidAttribute("678c0b04-7d66-11d2-a67d-00105a42887f")
        // CapeValidationStatus_IID),
]
public enum class CapeValidationStatus {
        CAPE_NOT_VALIDATED = 0,
        CAPE_INVALID = 1,
        CAPE_VALID = 2
};
```

4.1.2 Defining Interfaces

A COM type library is basically compiled idl code, so the type library will be considered as its underlying idl code. Further, a similar translation of the interface definition will be performed by the type library importer, and implementing classes will need to use the .NET style-function signatures as opposed to the COM idl function signatures. As an example, the *ICapeUnit* interface will be considered here as idl, and below as a .NET interface. The COM idl code for the *ICapeUnit* interface is:

```
//      This interface provides the basic functionality for a Unit
//      Operation component
[
        object,
        uuid(ICapeUnit_IID),
        dual,
        helpstring("ICapeUnit Interface"),
        pointer_default(unique)
]
interface ICapeUnit : IDispatch
{
        //      Get the collection of unit operation ports
        //
        //  CAPE-OPEN exceptions:
        //  ECapeUnknown,  ECapeFailedInitialisation,  ECapeBadInvOrder
        [propget, id(1), helpstring("Gets the whole list of ports")]
        HRESULT ports([out, retval] CapeInterface* ports);

        // Gets the flag to indicate unit's validation status
        //· notValidated(0),invalid(1) or valid(2)
        //
        // CAPE-OPEN exceptions
        // ECapeUnknown, ECapeInvalidArgument
        [propget, id(2), helpstring("Get the unit's validation status")]
        HRESULT ValStatus([out, retval] CapeValidationStatus *valStatus);

        //      Executes the necessary calculations involved in the unit
        //      operation model
        //
```

```
        //  CAPE-OPEN exceptions raised:
        //  ECapeUnknown, ECapeBadInvOrder, ECapeOutOfResources,
        //  ECapeTimeOut, ECapeSolvingError, ECapeLicenceError
        [id(3), helpstring("Performs unit calculations")]
        HRESULT Calculate();

        //     Validate that the parameters and ports are all valid
        //
        //  CAPE-OPEN exceptions:
        //  ECapeUnknown,  ECapeBadCOParameter,  ECapeBadInvOrder
        [id(4), helpstring("Validate the Unit")]
        HRESULT Validate([ACTUALLYout] CapeString* message,
                [out, retval] CapeBoolean* isValid);
};
```

A GUID assigned to the interface (`ICapeUnit_IID`), which is assigned a value using a #define compiler pragma. This interface derives from *IDispatch*, which means that it supports late binding. The function headers are such that the string values are returned by reference and the actual return value for the function is an HRESULT, which is a 32-bit integer that indicates whether an error condition occurred during the function call. The attributes preceding the function definition have the following meanings:

- `Propput/Propget` indicates that this function either gets or sets a property value
- `id(1)` is the identification number of the function in the Dispatch interface
- `helpstring("…")` A help string that provides information about the function
- `out, retval` indicates that the variable is function's return value in Visual Basic and that the value will be returned by reference

By comparison, the same interface defined in C++/CLI is as follows:

```
//      This interface provides the basic functionality for a Unit
//      Operation component
//      CAPE-OPEN v1.0
[
        ComVisibleAttribute(true),
        Guid("678c0998-0100-11d2-a67d-00105a42887f"),//ICapeUnit_IID),
        System::ComponentModel::DescriptionAttribute("ICapeUnit Interface")
]
public interface class ICapeUnit
{
        //     Get the collection of unit operation ports
        //
        //  CAPE-OPEN exceptions:
        //  ECapeUnknown,  ECapeFailedInitialisation,  ECapeBadInvOrder
        [DispIdAttribute(1), System::ComponentModel::DescriptionAttribute
                                ("Gets the whole list of ports")]
        property Object^ ports
        {
                [returnvalue: MarshalAs(UnmanagedType::IDispatch)]
                Object^ get();
        };

        // Gets the flag to indicate unit's validation status
        //· notValidated(0),invalid(1) or valid(2)
        //
        // CAPE-OPEN exceptions
        // ECapeUnknown, ECapeInvalidArgument
        [DispIdAttribute(2), System::ComponentModel::DescriptionAttribute
                ("Get the unit's validation status")]
        property CapeValidationStatus ValStatus
        {
                CapeValidationStatus get();
        };
```

```
    //     Executes the necessary calculations involved in the unit
    //     operation model
    //
    //   CAPE-OPEN exceptions raised:
    //   ECapeUnknown,  ECapeBadInvOrder,  ECapeOutOfResources,  ECapeTimeOut,
    //   ECapeSolvingError,  ECapeLicenceError
    [DispIdAttribute(3), System::ComponentModel::DescriptionAttribute
                        ("Performs unit calculations")]
    void Calculate();

    //     Validate that the parameters and ports are all valid
    //
    //   CAPE-OPEN exceptions:
    //   ECapeUnknown,  ECapeBadCOParameter,  ECapeBadInvOrder
    [DispIdAttribute(4), System::ComponentModel::DescriptionAttribute
           ("Validate the Unit"),
           returnvalue: MarshalAs(UnmanagedType::VariantBool)]
    bool Validate(String^ %message);
};
```

This .NET-based *ICapeUnit* interface definition provides the same information as the COM idl above. The *COMVisible* attribute is required for COM interop. Without this attribute, the interface will not be made visible to COM interop. Further, classes deriving from this interface will also not be visible to COM, even if they are marked *COMVisible*. The *GUID* attribute is used to provide the COM interface identification GUID, and serves the same function as the COM idl *uuid* attribute. The GUID attribute should be used on all interfaces and classes exposed to COM otherwise, a GUID will be assigned to the object when it is registered in the system registry. *System.ComponentModel.Description* attribute provides similar functionality to the COM idl *helpstring* attribute. The *DispId* attribute serves the same function as the *id* attribute in the COM idl.

The most obvious difference in the above interface definitions are the function definitions themselves. In the case of the ports property, it is defined as a property with an *Object* return type. The other methods are defined so that the value marked with the *out, retval* attribute in the COM idl file are of the actual type of the return value for the function. Exceptions are thrown by the .NET class implementing this interface, so the return value for the function is not an *HRESULT*. As discussed below, a .NET exception can be assigned an *HRESULT* which is returned to COM when the exception is thrown.

For the ports property, the *MarshalAs* attribute tells the interop marshaller that the value is actually an *IDispatch* type object. Without the *MarshalAs* attribute, the marshaller would make this return value a variant. The *MarshalAs* attribute is also used on the Validate method to inform the marshaller that the value is a Variant Boolean, not a C++ Boolean value.

4.1.3 Importing and Exporting Type Libraries

COM type libraries can be imported directly for use in .NET using the type library importer (tlbimp.exe). Basically, to use the currently defined CAPE-OPEN type libraries, one simply need to import the CAPE-OPEN version 1.0 Component Object Model (COM) type library into a .NET Primary Interop Assembly (PIA) (Troelsen 2002). The type library can be signed with a key file, which results in a strongly-named PIA. This process will create a *CAPEOPEN100* namespace that contains all the CAPE-OPENv1.0 interface definitions and provided a built-in marshalling of data from the CAPE-OPEN COM objects to the .NET framework using the data types indicated above. Use of the PIA would allow CAPE-OPEN compliant .NET based objects to be used in a COM-based flowsheeting tool directly. This route provides the quickest mechanism to provide a .NET interop assembly.

Alternatively, as shown above, the interfaces can be defined in a .NET language and exported to COM via the type library exporter (*tlbexp.exe*) facility. In this case, the interfaces will be created using .NET file signatures. Each interface can be assigned a GUID that COM can use to identify the interface. During the process of exporting the type library, the function signatures are converted to the format found in the idl files. By assigning .NET-based interfaces the same GUID as the matching COM interface, .NET interfaces with the same functions as the COM interfaces can continue to be used. Care must be taken in this approach

to ensure that the proper *MarshalAs* attributes are applied to variables and return values primarily to ensure that objects are returned as a dispatch interface or that Boolean values are converted to variant Boolean values.

4.2 Error Handling

COM provides an error handling API that uses an IErrorInfo interface and the *GetErrorInfo* API to transmit exceptions from the object that raises them to the container application. However, because of difficulties associated with propagating errors using this API to previous versions of Visual Basic, the CAPE-OPEN Error Common Interface standard chose to require all objects that support the Error Common Interface to implement all error interfaces that they are able to raise (CO-LaN 2003). As a result of the deviation of the CAPE-OPEN error handling specification from the COM *GetErrorInfo* API, .NET-based objects must deviate from using .NET exceptions as the sole mechanism to transmit and receive error information in order to fully interoperate using the CAPE-OPEN error interfaces. This section describes implementation of PMCs and PMEs to enable error handling in compliance with the CAPE-OPEN standards. An appendix to this document presents an implementation of the .NET error handling mechanisms and discusses the effect of the use of standard .NET handling mechanisms on CAPE-OPEN compliance.

.NET allows the use of structured exceptions similar to CORBA and C++. Structured exception can be thrown by an object when an error condition occurs during a method call and the calling object can catch the exception and handle the error causing condition. Use of exceptions is in contrast with the error handling mechanisms of COM which uses a 32-bit integer error code, called an HRESULT, as a return for the function call when error conditions are encountered. The advantage of using structured exceptions is that information such as a message and the source of the exception can be included in the exception. This section will discuss handling an HRESULT returned by a COM method and the implementation of an application exception class that can be thrown when computation exception, such as dividing by zero, occurs in a function.

4.2.1 Exception Handling for a COM-Based PMC in a .NET-based PME

When an error occurs in a CAPE-OPEN COM-based PMC, the current function returns an HRESULT value, that COM interop automatically recognizes as an exception and throws a *System.Runtime.InteropServices. COMException* object. The HRESULT for the exception is the same as the COM HRESULT returned by the function call. In order to comply with the CAPE-OPEN error handling mechanisms, the object throwing the exception must expose the appropriate CAPE-OPEN error interface(s) to access information about the exception. When a COM-based PMC returns an error HRESULT, the runtime callable wrapper then generates a .NET *COMException* which would then be caught by the PME.. The PME will then cast the PMC to the appropriate CAPE-OPEN error interface to obtain detailed error information. If the COM-based PME is contained in a wrapper class, the wrapper could then generate and throw a .NET based exception as described in the appendix.

Table 2. Common HRESULT Error Values and Corresponding .NET Exceptions

HRESULT	.NET exception
MSEE_E_APPDOMAINUNLOADED	AppDomainUnloadedException
COR_E_APPLICATION	ApplicationException
COR_E_ARGUMENT or E_INVALIDARG	ArgumentException
COR_E_DIVIDEBYZERO	DivideByZeroException
COR_E_INDEXOUTOFRANGE	IndexOutOfRangeException
COR_E_IO	IOException

COR_E_SECURITY	SecurityException
COR_E_SERIALIZATION	SerializationException
COR_E_STACKOVERFLOW,ERROR_STACK_OVERFLOW	StackOverflowException
COR_E_SYSTEM	SystemException
Unidentified HRESULT	COMException

The following code sample shows how toihandle an exception from a COM PMC in a CAPE-OPEN compliant manner. The code might be part of the calculation routine of a unit operation PMC while setting a property in a material object (variable *mo*) connected to one of its ports. Exceptions defined in the above table are caught by their respective .NET exception class. All errors defined in the CAPE-OPEN error handling specification are caught through the *COMException* block and are then further distinguished by the HResult code which can be read from the *ErrorCode* attribute of the COMException object.

```
virtual Calculate()
{
	// …
	try
	{
		mo.SetProp( "enthalpy", "liquid", empty, "mixture", "moles", val);
	}
	catch( DivideByZeroException dbz )
	{
		// …
	}
	// … handle other exceptions from the above table where necessary
	catch( COMException ex )
	{
		// handle all errors of ICapeThermoMaterialObject.SetProp
		switch( ex.ErrorCode )
		{
			case (int)CapeErrorInterfaceHR.ECapeBadArgumentHR:
			{
				ECapeUser ecu = mo as ECapeUser;
				// use ECapeUser members such as ecu.code, ecu.description, …

				ECapeBadArgument ecba = mo as ECapeBadArgument;
				// access ECapeBadArgument member ecba.position
			} break;
			// …
		}
	}
}
```

It should be noted, that the above code does not check whether the material object actually supports the interfaces ECapeUser and ECapeBadArgument as described by the error handling interface specification.

4.2.2 Throwing Exceptions from a .NET-Based PMC in a COM-based PME

In .NET, the application-based structured exception classes should derive from the .NET application exception class, *System.ApplicationException* (DeRemer 2004) which can communicate an HResult value to the calling COM component As indicate above, CAPE_OPEN error handling differs from the standard COM GetErrorInfo API. This sections describes the mechanism required for objects to report error information using the CAPE-OPEN error handling standards.

The CAPE-OPEN objects should implement the error interfaces and return the appropriate CAPE-OPEN defined HRESULT values (CO-LaN 2003). This approach requires two steps for implementation in .NET.

First, an exception must be defined that contains a proper HResult value. Second, the component must implement the error interfaces defined in the error handling specification.

A straight forward approach to define exception classes with HResults is to include the HResult definition into the constructor of the exception class:

```
public class CapeBadArgumentException : CapeDataException
{
	public CapeBadArgumentException()
	{
		this.HResult = (int)CapeErrorInterfaceHR.ECapeBadArgumentHR;
	}
}
```

Raising this exception within some .NET-based code will transmit the defined HResult value to the calling COM component. The caller will then try to evaluate the CAPE-OPEN error interfaces in order to find out further information on the error. A possible implementation of some of these interfaces might look like this:

```
public class PropertyPackage : ICapeIdentification, ICapeThermoPropertyPackage,
                                  ECapeUser, ECapeRoot, ECapeBadArgument // , …
{
	// ECapeRoot members
	string m_name;
	public string name
	{
		get { return m_name; }
	}

	// ECapeUser members
	string m_description, m_scope, m_more_info, m_interface_name, m_operation;
	int m_code;
	public string description
	{
		get { return m_description; }
	}
	// other members for ECapeUser

	// same for other error interfaces

	// remainder of implementation
}
```

The last step in the process is the actual throwing of the exception by the source component. In this case, the desired exception to be thrown is a bad argument exception. First, all members corresponding to interfaces relevant for ECapeBadArgument must be set, finally the actual exception is thrown:

```
public void CalcProp( … )
{
	// …

	// raise an exception, first set all members for relevant error interfaces
	m_name = "Bad argument encountered";
	m_description = "The phase argument is not supported";
	// set other error members …

	// finally, raise the exception
	throw new CapeBadArgumentException();

	// …
}
```

The above error handling code is used in a CAPE-OPEN compliant property package and was tested against the CAPE-OPEN based flowsheeting environment COFE (Amsterchem, http://www.amsterchem.com/) and error information was successfully transmitted, including detail information about the errors indicated.

4.3 Implementation of Collections

Collection implementations need to be able to implement the CAPE-OPEN collection interface, allow COM-based clients to enumerate the collection and provide .NET clients access to collection members as well. .NET framework Version 1.x has two basic collection classes available, an *Array* and an *ArrayList*. Version 2.0 of the .NET framework adds a number of generic collection classes that can also be used. The advantage to the generic collections in .NET v2.0 is the fact that the contained values are strongly typed, whereas the arrays in .NET 1.x are not. That means that a .NET 2.0 collection of parameters could be defined at design time to only contain elements derived from *ICapeParameter*. In either case, the .NET collections can be wrapped with a class that exposes the CAPE-OPEN *ICapeCollection* interface. It should be noted that the CAPE-OPEN tester uses a 1-indexed array, that is, the lowest value of the index sent is 1, as such the *Item* method needs to subtract 1 from the index requested as the .NET collections are 0-indexed. A .NET version 2.0 parameter collection could be defined as follows:

```
[
	Serializable,
	ComVisibleAttribute(true),
	GuidAttribute("6870DFAF-746E-4dfb-A26F-1261DE7B17EE")
]
public ref class CParameterCollection :  public BindingList<ICapeParameter^>,
	public ICapeIdentification,
	public ICapeCollection,
	public ICapeUnitCollection
{
private:
	String^ m_ComponentName;
	String^ m_ComponentDescription;

public:
	CParameterCollection(void)
	{
	};

	~CParameterCollection ()
	{
		this->Clear();
	}

	// ICapeIdentification methods
	virtual property String^ ComponentName{
		String^ get()
		{
			return m_ComponentName;
		}
		void set (String^ value)
		{
			m_ComponentName = value;
		}
	}

	virtual property String^ ComponentDescription{
		String^ get()
		{
			return m_ComponentDescription;
		}
			void set (String^ value)
		{
			m_ComponentDescription = value;
		}
	}


	//These are the ICapeCollection member implementations
	virtual int CollectionCount() = ICapeCollection::Count
	{
		return this->Items->Count;
	}
```

```
        [returnvalue: MarshalAs(UnmanagedType::IDispatch)]
        virtual Object^ CollectionItem(Object^ index) = CapeOpen::ICapeCollection::Item
        {
                Object^ retval = nullptr;
                Type^ indexType = index->GetType();
                if ((indexType == System::Int16::typeid) ||
                (indexType == System::Int32::typeid)
                || (indexType == System::Int64::typeid)){
                        int i = dynamic_cast<IConvertible^>(index)->ToInt32(
                                                gcnew NumberFormatInfo());
                        retval = this[i-1];
                }
                if ((indexType == System::String::typeid)){
                        String^ name = dynamic_cast<String^>(index);
                        for (int i = 0; i < this->Count; i++){
                                ICapeIdentification^ p_Id;
                                p_Id = dynamic_cast<ICapeIdentification^>(this[i]);
                                if (!String::Compare(p_Id->ComponentName, name)){
                                        retval = this[i];
                                }
                        }
                }
                return retval;
        }
};
```

One other key issue in the *Item()* method is the check to determine if the index is either a 16-bit, 32-bit, or 64-bit integer. The CAPE-OPEN tester actually sends both 16-bit and 32-bit integer indices to the Item function. It is not obvious from the tester's Visual Basic 6 code as to whether the value was a 16-bit or 32-bit integer, so in order to prevent the *Item* method from failing as a result of the client application, the index is tested to see if it is any integer type available to .NET.

4.4 Persistence

There are three persistence scenarios that need to be considered here, persistence of a .NET-based PMC in a COM-based PME, persistence of a COM-based PMC in a .NET-based PME, and persistence of a .NET-based PMC in a .NET based PME.

4.4.1 Persistence of a .NET-based PMC in a COM-based PME

- General Procedure

CAPE-OPEN compliant simulators (process modelling environments) expect components such as unit operations to implement an interface called *IPersistStreamInit*. However, this interface has no equivalent in the .NET framework and is not taken care of by the automatically generated wrappers. Hence, what needs to be done is to emulate an interface with identical method names and parameters.

- Defining Required Interfaces

The interface *IPersist* is an interface from which other persistence interfaces inherit. It must be defined first (all code samples here are in C#):

```
[
                InterfaceType(ComInterfaceType::InterfaceIsIUnknown),
                Guid("0000010c-0000-0000-C000-000000000046"),
                ComVisibleAttribute(true)
        ]
        public interface class IPersist
        {
                void (out Guid pClassID);
        };
```

```
[
	InterfaceType(ComInterfaceType::InterfaceIsIUnknown),
	Guid("00000109-0000-0000-C000-000000000046"),
	ComVisibleAttribute(true)
]
public interface class IPersistStream : IPersist
{
	void GetClassID(out Guid pClassID);
	[PreserveSig]
	int IsDirty( );
	void Load(System::Runtime.InteropServices.ComTypes.IStream pStm);
	void Save(System::Runtime.InteropServices.ComTypes.IStream pStm,
		     [In, MarshalAs(UnmanagedType.Bool)] bool fClearDirty);
	void GetSizeMax(long %pcbSize);
};
[
	InterfaceType(ComInterfaceType::InterfaceIsIUnknown),
	Guid("7FD52380-4E07-101B-AE2D-08002B2EC713"),
	ComVisibleAttribute(true)
]
public interface class IPersistStreamInit : IPersist
{
	void GetClassID(out Guid pClassID);
	[PreserveSig]
	int IsDirty( );
	void Load(System.Runtime.InteropServices.ComTypes.IStream pStm);
	void Save(System.Runtime.InteropServices.ComTypes.IStream pStm,
		     [In, MarshalAs(UnmanagedType.Bool)] bool fClearDirty);
	void GetSizeMax(long %pcbSize);
	void InitNew();
};
```

- Implementing *IPersistStreamInit*

Now, you can implement persistence on behalf of your component by fulfilling the *IPersistStreamInit* interface. Arbitrary data (such as a string) can be given to the *UCOMIStream* interface (in Visual Studio 2005, and the .NET Framework version 2.0, the *UCOMIStream* interface has been replaced with the *System.Runtime.InteropServices.ComTypes.IStream* interface) in the *Save* method and is retrieved in the *Load()* method. Implementing *GetSizeMax()* is not strictly required. An example *Save()* method which first writes the amount of data and the an actual object in binary form looks like this:

```
public void Save(UCOMIStream pStm, bool fClearDirty)
 {
     int cb;
     int* pcb = &cb;
     byte[] arrLen = new byte[2];
     // Convert the string into a byte array
     MemoryStream memoryStream = new MemoryStream();
     BinaryFormatter binaryFormatter = new BinaryFormatter();
     binaryFormatter.Serialize(memoryStream, data);
     byte[] bytes = memoryStream.ToArray();
     memoryStream.Close();
     // construct length (separate into two separate bytes)
     arrLen[0] = (byte)(bytes.Length % 256);
     arrLen[1] = (byte)(bytes.Length / 256);
     // Save the array in the stream
     stream.Write(arrLen, 2, new IntPtr(pcb));
     stream.Write(bytes, bytes.Length, new IntPtr(pcb));
 }
```

And a corresponding *Load()* method reading an object from a stream looks like this:

```
 public void Load(UCOMIStream pStm)
 {
     object data = null;
     int cb;
```

```
    byte[] arrLen = new Byte[2];
    // Read the length of the string
    int* pcb = &cb;
    stream.Read(arrLen, 2, new IntPtr(pcb));
    // Calculate the length
    cb = 256 * arrLen[1] + arrLen[0];
    // Read the stream to get the string
    byte[] bytes = new byte[cb];
    stream.Read(bytes, cb, new IntPtr(pcb));
    // Deserialize byte array
    MemoryStream memoryStream = new MemoryStream(bytes);
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    object objectDeserialize = binaryFormatter.Deserialize(memoryStream);
    if (objectDeserialize != null)
    {
        data = objectDeserialize;
    }
    memoryStream.Close();
    return data;
}
```

The implementation of the other method in the interface is trivial.

4.4.2 Persistence of a .NET-based PMC in a .NET-based PME

Of the three cases, this is the simplest, use .NET-based object serialization. In this case, all PMCs need to implement the *ISerializable* interface and have the *Serializable* attribute applied. As long as all objects referenced by the PMC are serializable, then the PMC can be serialized by the PME using basic .NET serialization procedures. It is important to note that the *Serializable* attribute is not inheritable. All PMCs, PMC based classes, and member variables referenced by the PMC, must be serializable. In the event that all objects are not serializable, the PMC can mark the non-serializable objects as such and implement custom serialization, as described in MSDN.

4.4.3 Persistence of a COM-based PMC in a .NET-based PME

The simplest method to persist COM-based PMCs in a .NET PME is to create a .NET-based wrapper class for the COM PMC that is serializable. The wrapper needs to test the COM PMC to determine which COM persistence mechanism it uses and then call the appropriate functions to persist the object to a serializable stream. This will require use of custom serialization and implementation of the *ISerializable* interface. The COM-based PMC can then be serialized to a class that wraps a *System.IO.MemoryStream* object and implements the *System.Runtime.InteropServices.ComTypes.IStream* interface (Nathan 2002).

4.5 Registering Objects

The .NET object must be registered and placed in the appropriate CAPE-OPEN component categories. This can be accomplished by instructing Visual Studio to register the class library for COM interoperation. This process adds the classes to the .NET component category in the system registry. In order to expose the object as a CAPE-OPEN-based PMC, the component must also be registered in the appropriate CAPE-OPEN categories, which must be accomplished using a COM registration function that creates the appropriate CAPE-OPEN categories and adds the object to the categories (Troelsen 2002). The following code snippet implements the COM registration and unregistration functions for a unit operation, and populates the Cape description key using information contained in the assemblies custom attributes.

```
[ComRegisterFunctionAttribute]
static void RegisterFunction(Type^ t)
{
      String^ keyname = String::Concat(L"CLSID\\{", t->GUID.ToString(),
            L"}\\Implemented Categories");
      Microsoft::Win32::RegistryKey^ key = Win32::Registry::ClassesRoot->
```

```
        OpenSubKey(keyname, true);
key->CreateSubKey("{678c09a5-7d66-11d2-a67d-00105a42887f}");//UnitOp CATID
key->CreateSubKey("{678C09A1-7D66-11D2-A67D-00105A42887F}");//CapeObject
key->Close();
key = Win32::Registry::ClassesRoot->OpenSubKey(String::Concat(L"CLSID\\{",
         t->GUID.ToString(), L"}"), true);
key->CreateSubKey(L"CapeDescription");
key->Close();
key = Microsoft::Win32::Registry::ClassesRoot->OpenSubKey(
        String::Concat(L"CLSID\\{", t->GUID.ToString(), L"}",
        L"\\CapeDescription"), true);
Assembly^ assembly = Assembly::GetExecutingAssembly();
if (assembly && String::Compare(assembly->GetName()->Name, L"mscorlib")){
        //here we are parsing out copyright info attribute
        array<Object^>^ CopyrightInfo = assembly->GetCustomAttributes(
                AssemblyCopyrightAttribute::typeid,false)
        //here we are parsing out copyright info attribute
        array<Object^>^ DescriptionInfo = assembly->GetCustomAttributes(
                AssemblyDescriptionAttribute::typeid,false);
        //here we are parsing out copyright info attribute
        array<Object^>^ VersionInfo = assembly->GetCustomAttributes(
                AssemblyFileVersionAttribute::typeid,false);
        //here we are parsing out copyright info attribute
        array<Object^>^ CompanyURLInfo = assembly->GetCustomAttributes(
                AssemblyProductAttribute::typeid,false);
        String^ CopyRightInfoString = "";
        for(int j=0; j<CopyrightInfo->Length; j++ )
        {
                if(j>0)
                {
                        CopyRightInfoString += CopyRightInfoString + " ," +
                                (dynamic_cast< AssemblyCopyrightAttribute^>(
                                CopyrightInfo[j]))->Copyright;
                }
                else
                {
                        CopyRightInfoString = (dynamic_cast<AssemblyCopyrightAttribute^>
                                (CopyrightInfo[j]))->Copyright;
                }
        }
        //here we are parsing out title info attribute
        array<Object^>^ TitleInfo = assembly->GetCustomAttributes(
                AssemblyTitleAttribute::typeid,false);
        String^ TitleInfoString = "";
        for(int j=0; j<TitleInfo->Length; j++)
        {
                if(j>0)
                {
                        TitleInfoString += TitleInfoString + " ," +
                                (dynamic_cast<AssemblyTitleAttribute^>(TitleInfo[j]))
                                ->Title;
                }
                else
                {
                        TitleInfoString = dynamic_cast< AssemblyTitleAttribute^>
                                (TitleInfo[j])->Title;
                }
        }
        //here we are parsing out company name attribute
        array<Object^>^ CompanyInfo = assembly->GetCustomAttributes(
                AssemblyCompanyAttribute::typeid, false);
        String^ CompanyInfoString = "";
        for(int j=0; j<CompanyInfo->Length; j++)
        {
                if(j>0)
                {
                        CompanyInfoString += CompanyInfoString + " ," +
                                (dynamic_cast< AssemblyCompanyAttribute^>(CompanyInfo[j]))
                                ->Company;
                }
                else
```

```
                {
                        CompanyInfoString = (dynamic_cast<AssemblyCompanyAttribute^>
                                (CompanyInfo[j]))->Company;
                }
        }
        String^ descriptionInfoString = L"";
        for(int j=0; j<DescriptionInfo->Length; j++)
        {
                if(j>0)
                {
                        descriptionInfoString += descriptionInfoString + " ," +
                          dynamic_cast< AssemblyDescriptionAttribute^>(DescriptionInfo[j])
                          ->Description;
                }
                else
                {
                        descriptionInfoString = dynamic_cast< AssemblyDescriptionAttribute^>
                                (DescriptionInfo[j])->Description;
                }
        }
        String^ versionInfoString = L"";
        for(int j=0; j<VersionInfo->Length; j++)
        {
                if(j>0)
                {
                        versionInfoString += versionInfoString + " ," +
                                dynamic_cast< AssemblyFileVersionAttribute^>(VersionInfo[j])
                                ->Version;
                }
                else
                {
                        versionInfoString = dynamic_cast<AssemblyFileVersionAttribute^>
                                (VersionInfo[j])->Version;
                }
        }
        String^ companyURLInfoString = L"";
        for(int j=0; j<CompanyURLInfo->Length; j++)
        {
                if(j>0)
                {
                        companyURLInfoString += companyURLInfoString + " ," +
                                dynamic_cast< AssemblyProductAttribute^>(CompanyURLInfo[j])
                                ->Product;
                }
                else
                {
                        companyURLInfoString = dynamic_cast< AssemblyProductAttribute^>
                                (CompanyURLInfo[j])->Product;
                }
                key->SetValue("Name", TitleInfoString);
                key->SetValue("Description", descriptionInfoString);
                key->SetValue("CapeVersion", "1.0");// name
                key->SetValue("ComponentVersion", versionInfoString);// name
                key->SetValue("VendorURL", CompanyInfoString);// name
                key->SetValue("HelpURL", CompanyInfoString);// name
                key->SetValue("About", CopyRightInfoString);// name
                key->Close();
        }
    }
}

[ComUnregisterFunctionAttribute]
static void UnregisterFunction(Type^ t)
{
    String^ keyname = String::Concat(L"CLSID\\{", t->GUID.ToString(),
        L"}\\Implemented Categories\\{678c09a5-7d66-11d2-a67d-
        00105a42887f}");
    Microsoft::Win32::Registry::ClassesRoot->DeleteSubKey(keyname);
    keyname = String::Concat(L"CLSID\\{", t->GUID.ToString(), L"}\\Implemented
        Categories\\{678C09A1-7D66-11D2-A67D-00105A42887F}");
    Microsoft::Win32::Registry::ClassesRoot->DeleteSubKey(keyname);
```

```
        keyname = String::Concat(L"CLSID\\{", t->GUID.ToString(),
                L"}\\CapeDescription");
        Microsoft::Win32::Registry::ClassesRoot->DeleteSubKey(keyname);
}
```

The remaining issue is the instantiation of the object by COM. This is not a trivial task as .NET registers that the executable for this registry item as the .NET framework core library, mscorlib.dll. In order to instantiate the object, the assembly that contains it must be placed in one of three a specific locations. These locations are 1) the global assembly cache, 2) a code base which is specified in the registry key, or 3) a directory located within the directory that contains the application attempting to instantiate the object (Troelsen 2002). By selecting the register for COM interop compiler option in C# and Visual Basic, the code base registry key is properly configured for COM interop.

It should be noted that the latest release of Visual Studio promises registration-free COM interoperation. Registration-free COM interop uses an application manifest to provide the information required for class instantiation. To date, this has not been tested.

5. Motivation and Roadmap for .NET-based CAPE-OPEN

This section is intended to provide a discussion of the options available for moving forward with the development and ongoing extension of CAPE-OPEN interface standards given the development of the .NET framework by Microsoft. So far, this document has provided a brief discussion of the added features associated with .NET, and has shown that .NET objects can be readily used in a COM environment. Further, a .NET environment can also readily use objects created in COM. This demonstrated interoperability is the first step in moving from one object model to another – ensuring legacy objects are supported. Clearly, interoperability, legacy support, and added features are important in this endeavour, but other issues remain, such as whether there will be a requirement for future changes in object model and will the new object model be robust enough to evolve as new technologies are developed and brought to bear on future problems.

As a starting point for this discussion, it should be recalled that "The first objective of the [CAPE-OPEN] partnership was to understand how software for designing and optimising process plants could be modified to make use more cost-effective by integrating software pieces one into another." (Pons 2003) At present, there stands significant experience demonstrating the success of this endeavour as COM-based PMCs can now readily be utilized in a wide range of PMEs through the use of the CAPE-OPEN interface set. While this effort is not complete, some interface packages require little more than fine tuning of tested interface models while other interface packages are still in their infancy, future efforts should build upon past accomplishment. A clear consideration is that any changes to the object model build upon this experience, and .NET meets this criterion.

One key issue related to the continued use of COM is that COM is a proprietary technology created by Microsoft, and Microsoft is in the process of phasing it out. Microsoft's COM web page (http://www.microsoft.com /com/default.mspx) clearly states: "Microsoft recommends that developers use the .NET Framework rather than COM for new development." At this point, it should be noted that the need to consider a new object model is due to the reliance on a previous proprietary model and that the .NET Framework and the Common Language Runtime (CLR) are Microsoft proprietary models. The risk that the .NET object model will be deprecated or made obsolete is reduced by the fact that the Common Language Infrastructure (CLI), the C# programming language, and the C++/CLI programming language are open standards that have been accepted by the European Computer Manufacturer's Association (ECMA). Microsoft has implemented a Shared Source version of the CLI and C# language available at http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en. Third-party implementations of both the CLI and C# language exist, such as the Mono Project (http://www.mono-project.com/Main_Page) and dotGNU (http://dotgnu.info/). These implementations can run on not just in the Windows environment, but also on Linux/UNIX and Apple's Macintosh Operating System. The fact that open-source, shared source, and third-party implementations of the CLI (and therefore, the .NET Framework) exist reduces the risk that a new object model will be designed that will supplant this effort.

Another area that must be considered is the ability of the object model to evolve as new technologies are brought to bear on CAPE-related problems. Recently, CAPE-OPEN-based process simulation tools have been demonstrated on a parallel processing system. Technologies that one can readily expect process simulation applications include advances in processor architecture and distributed applications. Microprocessor manufacturers and software developers are slowly moving away from 32-bit processors to 64-bit processors, which provide more addressable memory and faster computation. Grid computing (Foster, Kesselman et al. 2001) is focused on large scale resource sharing, innovative applications and high performance computation. Grid computing technology is reliant on flexible, secure, coordinated resource sharing amongst members of virtual organizations, which is likely to be required for issues such as supply chain management and production scheduling.

At present, the .NET Framework, and ultimately the standardized CLI, appear to meet the needs of providing a relatively stable development platform for the foreseeable future. This architecture improves on issues related to COM development such as registration (dll Hell) and security. Given the current state of the .NET Framework, and the third-party/cross platform implementations of the CLI, adoption of these technologies as

a replacement for COM is not only inevitable, but expedient. The CAPE-OPEN consortium will discuss the following possible course of actions:

1. All current (version 1.x) COM-based type libraries will be made available as Primary Interop Assemblies for use in .NET development. The conversion step is done automatically using the type library importer utility (tlbimp.exe) that is part of the Microsoft .NET SDK,

2. Interface definitions currently under development must include .NET-based interface definitions in C#, and COM idl/type libraries or a type library file exported from the .NET assembly.

3. A future version of CAPE-OPEN will consist of .NET-based assemblies, written in C#. The corresponding COM interface definitions will be exported from the .NET assemblies. The conversion step is done automatically using the type library exporter utility (tlbexp.exe) that is part of the Microsoft .NET SDK, This changes the main interface specification work from COM IDL to a .NET-based interface definition in C#.

4. All draft and final .NET-based assemblies will be strongly-named and will be formally published, with a unique *Assembly.FullName* property through the CAPE-OPEN Laboratories Network's web site, http://www.co-lan.org. In this step, it must be considered to what extent capabilities from .NET may be used that are not available via COM interoperability.

Interface definitions in CORBA have not been subject to discussion. Therefore, the list above does not include changes to the COLaN's position regarding CORBA.

6. Bibliography

Busby, S. and E. Jezierksi. (2001). "Microsoft .NET/COM Migration and Interoperability." from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp.

CAPE-OPEN (2000). Conceptual Design Document 2 (CDD2) for CAPE-OPEN Project, Global CAPE-OPEN.

CO-LaN (2003). Open Interface Specifications: Error Common Interface, Version 7, CAPE-OPEN Laboratories Network.

DeRemer, R. (2004). "Error Handling: Throwing Custom Exceptions from a Managed COM+ Server Application." MSDN Magazine **19**(3).

ECMA (2005). Standard Number 335, Common Language Infrastructure. Geneva, ECMA International.

Foster, I., C. Kesselman, et al. (2001). "The Anatomy of the Grid: Enabling Scalable Virtual Organization " The International Journal of High Performance Computing Applications **15**(3): 200-222.

Grimes, R. (2003). Programming with Managed Extensions for Microsoft Visual C++ .NET--Version 2003. Redmond, Microsoft Press.

Microsoft (1995). The Component Object Model Specification. Remond, Microsoft Corporation.

Nathan, A. (2002). .NET and COM: The Complete Interoperability Guide. Indianapolis, Que/Sams.

Pons, M. (2003). "Industrial Implementations of the CAPE-OPEN Standard." AIDIC Conference Series **6**: 253-262.

Troelsen, A. (2002). COM and .NET Interoperability. New York, Apress.

7. Appendix Creating and Throwing .NET Exceptions

This appendix describes the exception throwing mechanism used in .NET that would transmit error information back to COM. This method partially complies with the CAPE-OPEN standard, but in the event that an object throws an exception, the object itself does not expose the CAPE-OPEN error interfaces. Rather, in accordance with the COM API, the exception in obtained using the GetErrorInfo API. This error object would support the CAPE-OPEN error interfaces in addition to the COM *IErrorInfo* interface. Use of this mechanism would be preferred in a native .NET environment.

In .NET, the application-based structured exception classes should derive from the .NET application exception class, *System.ApplicationException* (DeRemer 2004). In order for the exceptions to be consistent with the error-handling protocol defined as part of CAPE-OPEN, the exceptions should implement the error interfaces and return the appropriate CAPE-OPEN defined HRESULT values (CO-LaN 2003).

In the current implementation of the CAPE-OPEN exception classes, all exception classes derive from a *CapeRoot* object that implements the *ECapeRoot* interface. As described above, one of the guidelines for custom exception classes used in an application framework is that the exception derive from the .NET *System.ApplicationException* class, and the second guideline states that the class name should end in Exception, as such the base CAPE-OPEN exception class should be called the *CapeRootException* class and expose the ECapeRoot interface. As a result, All CAPE-OPEN based exceptions thrown can be caught as either a *CapeRootException* or a *System.ApplicationException* in addition to being caught as the derived exception type. Further, the .NET-based exception classes expose an HRESULT value that is used like an HRESULT in COM. When a .NET exception is thrown by an object, a COM-based PME will receive the HRESULT specified in the CAPE-OPEN Error Handling Standard. An implementation of the *CapeRootException* class is shown below:

```
[
	Serializable,
	GuidAttribute("6727E5E4-16D0-4a88-9E4A-1607F179BC0B"),
	ComVisibleAttribute(true)
]
public ref class CapeRootException abstract:  public System::ApplicationException,
	public ECapeRoot
{
protected:
	String^ m_name;
	CapeRootException () : System::ApplicationException () {}
	CapeRootException (String^ message) : System::ApplicationException (message) {}
	CapeRootException (SerializationInfo^ info, StreamingContext context):
		 System::ApplicationException(info, context){}
	CapeRootException (String^ message, Exception^ inner) :
System::ApplicationException(message, inner){}

	public:
		// ECapeRoot method
		// returns the message string in the System::ApplicationException.
	virtual property String^ Name
	{
		String^ get(void)
		{
			return m_name;
		}
	}
};
```

As described in the CAPE-OPEN Error Handling document, the *CapeRootException* class is abstract. Derived classes will set the *m_name* member variable in their constructor so that the *ECapeRoot.Name* member can return the appropriate value. Further, the *System.Exception* class has four overloaded constructors which are duplicated in the *CapeRootException* class. The first constructor is a default and contains no additional information. The second constructor provides a string that is exposed as an error

message by the exception class. The third constructor is uses serialization information and a streaming context which enables the exception to be sent across application domains. The final constructor allows the inclusion of an inner exception, which will allow the thrower to include underlying exception information such as, if division by zero was attempted, the inner exception could indicate that in addition to, for example, a calculation error, providing more detail as to what failed and where.

The next layer in the CAPE-OPEN error handling standard is the *CapeUser* error. As described in the error handling standard, the *CapeUserException* class is also abstract, and it implements the *ECapeError* interface. The listing for the *CapeUserException* class is shown below. This class provides implementation of the *ECapeUser* interface through mapping the information requested by the *ECapeUser* interface to the equivalent information provided by the .NET base exception class.

```
[
	Serializable,
	GuidAttribute("AA381C62-E752-49a1-A6D2-BDD61D9177A4"),
	ComVisibleAttribute(true)
]
public ref class CapeUserException abstract:  public CapeRootException,
	public ECapeUser
{
protected:
	String^ m_interfaceName;

	CapeUserException () : CapeRootException ("CapeUserException"){}
	CapeUserException (String^ message) : CapeRootException (message) {}
	CapeUserException (SerializationInfo^ info, StreamingContext context):
CapeRootException(info, context){}
	CapeUserException (String^ message, Exception^ inner) : CapeRootException(message,
inner) {}

public:
	virtual property int code
	{
		int get (void)
		{
			return this->HResult;
		}
	}

	virtual property String^ description
	{
		String^ get (void)
		{
			return this->Message;
		}
	}

	virtual property String^ scope
	{
		String^ get (void)
		{
			return this->Source;
		}
	}

	virtual property String^ interfaceName
	{
		String^ get (void)
		{
			return m_interfaceName;
		}
	}

	virtual property String^ operation
	{
		String^ get (void)
		{
			return this->StackTrace;
```

```
            }
        }

        virtual property String^ moreInfo
        {
            String^ get (void)
            {
                return this->HelpLink;
            }
        }
};
```

The actual exception to be thrown when a computation error occurs is the *CapeComputationException* and it derives from the *CapeUserException* class, extending the *System.ApplicationException* class, and implementing the *ECapeRoot*, *ECapeUser*, and *ECapeComputation* interfaces. Again, the constructor calls the appropriate base class constructor with the arguments supplied and then calls the *Initialize()* method, which sets the *HResult* value of the exception to the proper value, the name of the exception class and the name of the interface exposed by the exception.

```
[
        Serializable,
        GuidAttribute("9D416BF5-B9E3-429a-B13A-222EE85A92A7"),
        ComVisibleAttribute(true)
]
public ref class CapeComputationException :  public CapeUserException,
        public ECapeComputation
{
        void Initialize(void)
        {
            this->HResult = (int)CapeErrorInterfaceHR::ECapeComputationHR;
            m_interfaceName = "ECapeComputation";
            m_name = "CapeComputationException";
        }
public:
        CapeComputationException () : CapeUserException (){Initialize();}
        CapeComputationException (String^ message) : CapeUserException (message)
        {Initialize();}
        CapeComputationException (SerializationInfo^ info, StreamingContext context):
            CapeUserException(info, context)
        {Initialize();}
        CapeComputationException (String^ message, Exception^ inner) :
            CapeUserException(message, inner)
        {Initialize();}
};
```

The last step in the process is the actual throwing of the exception by the source unit. In this case, the desired exception to be thrown is a divide by zero exception. A unit operation will be created that creates a variable of type integer and initializes it to zero. This value will then be divided into a number to create a *System.DivideByZeroException* that will be caught in the *try…catch* block. When the exception is caught, it will be re-thrown as the inner exception in a new *CapeComputationException*. A message will also be added to indicate the nature of the exception.

The final interoperability check is to place this unit operation in either the CAPE-OPEN tester or a CAPE-OPEN based flowsheeting environment such as COFE (Amsterchem, http://www.amsterchem.com/) and see what happens. Unfortunately, neither of these environments provides a detailed description of the error, but both indicate that an error occurred during calculation. The CAPE-OPEN tester simply states that an error occurred during calculation, whereas COFE indicated that the error was a computation error.