Integrating ECMA/ISO PCTE and OMG's CORBA

A Thesis Submitted for the Degree of Master of Science

by

Patricia Tangney B.Sc.

School of Computer Applications,
Dublin City University,
Ireland.

August 1995

Supervisor: Dr. John Murphy

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:______________________________ **Date:**_______________

Patricia Tangney

ACKNOWLEDGEMENTS

None of this would have been possible with the support and encouragement of my parents, Kathleen and Noel Tangney, I am forever in their debt.

I would like to express my sincere thanks to my supervisors John Murphy and Robert Cochran for their guidance and advice throughout this work, and to the Centre for Software Engineering, Dublin City University for sponsoring this research.

Many people have gracious with their time and help to me during the course of this research, I would like to acknowledge them all, but especially Johnathan Jowett and Ariela Stern for their patience and help.

Finally I would like to thank Patricia Magee and Denise Tangney for their help, and especially for providing the entertainment,

Buíochas le Dia,

Go raibh maith agaibh go léir.

Patricia Tangney

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

1.1 RESEARCH PURPOSE 2
1.2 PCTE AND OMA CONVERGENCE 4
1.3 BACKGROUND 5
1.3.1 OMG & CORBA 7
1.3.2 PCTE 7
1.4 TECHNICAL ISSUES 8
1.5 OVERVIEW 10

CHAPTER 2 PCTE 12

2.1 PCTE - THE STANDARD 12
2.2 ARCHITECTURE OF PCTE-BASED SEES 14
2.3 PCTE'S OBJECT MANAGEMENT SYSTEM 18
2.3.1 PCTE OBJECTS 19
2.3.2 LINKS 20
2.3.3 PCTE OPERATIONS 20
2.3.4 PCTE TYPES 21
2.4 PCTE'S DATA DEFINITION LANGUAGE 23
2.4.1 TYPE IMPORTATION DECLARATION 25
2.4.2 OBJECT TYPE DECLARATION 26
2.4.3 LINK TYPE DECLARATION 26
2.4.4 LINK TYPE EXTENSION 27
2.4.5 OBJECT TYPE EXTENSION 27
2.4.6 ATTRIBUTE TYPE DECLARATION 28
2.5 PCTE PROCESSES 29
2.5.1 INTERPROCESS COMMUNICATION 32
2.6 PCTE ACTIVITIES 33
2.7 IMPLEMENTATIONS 35
2.8 EVALUATION 36

CHAPTER 3 OMG CORBA 38

3.1 DISTRIBUTED COMPUTING 38
3.2 OBJECT MANAGEMENT GROUP 39
3.3 OBJECT MANAGEMENT ARCHITECTURE 41
3.3.1 OBJECT SERVICES 44
3.4 CORBA 46
3.4.1 STRUCTURE OF AN OBJECT REQUEST BROKER 47
3.4.2 CLIENT 49
3.4.3 OBJECT IMPLEMENTATIONS 49
3.4.4 OBJECT ADAPTER 50
3.5 INTERFACE DEFINITION LANGUAGE 51
3.5.1 INTERFACE DEFINITION 52
3.5.2 OPERATION DEFINITION 54
3.5.3 ATTRIBUTE DEFINITION 55

3.5.4 Enum & Type Declaration 56
3.5.5 Constant Definition 56
3.5.6 Exception Declaration 57
3.6 Implementations & Industrial Relevance 57
3.7 Conclusion 60

CHAPTER 4 INTEGRATING PCTE AND CORBA 61

4.1 Relationship of PCTE and OMA 62
4.1.1 Primary Features and Strengths of PCTE 63
4.1.2 Primary Features and Strengths of OMA 64
4.1.3 Complementary Standards 65
4.2 INTEGRATION STRATEGIES 67
4.3 Mapping DDL to IDL 71
4.4 MAPPING IDL TO DDL 73
4.5 IDL Interfaces for PCTE Tools 74
4.6 Related Works 76
4.6.1 PCIS 76
4.6.2 COHESIONworX/PCTE 78
4.6.3 OOTIS Tool Integration Model 80
4.7 Evaluation 81

CHAPTER 5 LIMITATIONS OF THE MAPPING OF DDL TO IDL 83

5.1 GENERAL MAPPING CONCEPTS 84
5.2 MAPPING DDL CONSTRUCTS TO IDL CONSTRUCTS 86
5.2.1 Mapping Type Importation Declarations 87
5.2.2 Mapping Attribute Type Declarations 87
5.2.3 Mapping Object Type Declarations 89
5.2.4 Mapping Object Type Extension Declarations 90
5.2.5 Mapping Link Type Declarations 91
5.2.6 Mapping Link Type Extension Declarations 93
5.3 LIMITATIONS OF THE MAPPING 95
5.4 EXTENDING DDL FOR COMPATIBILITY WITH IDL 98
5.5 EVALUATION 100

CHAPTER 6 IDL INTERFACES FOR PCTE TOOLS 102

6.1 General Concepts 103
6.2 A PCTE Tool's IDL interface 104
6.3 Implementing a PCTE tool's IDL interface 106
6.3.1 Esh Scripts 109
6.4 Tool Composition 111
6.5 Evaluation 112

CHAPTER 7 CONCLUSIONS 115

7.1 PCTE 116
7.2 CORBA 117

7.3 INTEGRATION STRATEGIES 118
7.3.1 DDL TO IDL 119
7.3.2 IDL TO DDL 120
7.3.3 IDL INTERFACES FOR PCTE TOOLS 120
7.4 FUTURE WORK 121
7.5 OVERALL CONCLUSIONS 123

BIBLIOGRAPHY 125

APPENDIX A INTERFACE DEFINITION LANGUAGE (IDL) 131

APPENDIX B DATA DEFINITION LANGUAGE (DDL) 140

APPENDIX C C PROG SDS 148

APPENDIX D EXAMPLE IDL INTERFACE FOR PCTE TOOLS 167

Integrating ECMA/ISO PCTE and OMA

By

Patricia Tangney

Abstract

The relationship between the Portable Common Tool Environment (PCTE) and the Object Management Group's Object Management Architecture (OMA) including the Common Object Request Broker Architecture (CORBA) specification can be viewed as a complementary one. The PCTE specification addresses the area of large to medium grain data integration for distributed Computer Aided Software Engineering environments. OMA is a set of specifications designed to promote interoperability between independently developed applications across distributed possibly heterogeneous environments based on the object oriented paradigm. CORBA is the communications heart of OMA. implicitly defining "distributed and secure execution and interprocess communication services".

The current PCTE standard is largely object oriented. However it is not fully object oriented because it does not define behaviour for PCTE objects. By using OMA to provide object behaviour for PCTE objects as well as making them fully object oriented greater control integration between PCTE objects could also be achieved. PCTE's Object Management System has a rich data modelling mechanism because it was designed to integrate complex data and relationships, therefore being suitable for use as a persistent store for OMA objects. Thus the convergence of PCTE and OMA into a single standard is attractive; work is currently underway on this by the OMG PCTE Special Interest Group.

However it will be sometime before a specification converging PCTE and CORBA is available. The purpose of this thesis is to find an interim integration strategy which can be used while waiting for their eventual convergence, since both specifications have much to offer each other. This thesis discusses the language mapping of DDL to IDL (and vice versa) and the definition of IDL interfaces for PCTE tools as strategies for the interim integration of PCTE and CORBA.

CHAPTER 1 INTRODUCTION

Recent trends in the computer industry have motivated the research which is the subject of this thesis. These trends include the shift away from mainframe systems to distributed computing, the popularity of the object oriented approach to software construction, strides towards the automation of software construction (Computer-Aided Software Engineering) or at least particular aspects of the software development process (e.g. coding or design tools), and the demand for greater software interoperability. The focus of this thesis is on the integration of two standards, Object Management Group's Object Management Architecture (OMA) and the Portable Common Tool Environment (PCTE), both of which have emerged because of these trends.

OMA is a set of specifications (including the Common Object Request Broker Architecture, CORBA) that are designed to "enable distributed integrated applications using an object oriented approach" [5]. The PCTE specification is primarily designed to address the integration of distributed CASE environments. Because OMA/CORBA and PCTE take different approaches to achieving integration, instead of overlapping, we discover they complement each other. Much work is being done by the OMG PCTE SIG (Special Interest Group) on the long term merging of both of these specification; currently specifications for a PCTE specification incorporating CORBA are being drafted. However the convergence of PCTE and OMA into a single standard is sometime in future. We believe the benefits of their integration are such that they warrant the development of a strategy which would allow the complementary integration of the current PCTE and OMA specifications, to be used while waiting for their eventual convergence into a single standard. Therefore the objective of this thesis is to achieve a short term integration of the current specifications of PCTE and OMA so that they may be used together now in a mutually beneficial manner.

1.1 Research Purpose

A wide range of Computer-Aided Software Engineering (CASE) tools are now available on the market, the purpose of which is to automate the aspects of the software development process. Until recently software has been developed predominantly on large centralised computer systems using a collection of tools bearing little or no relationship to one another. Although such isolated CASE tools did have the potential to reduce the production cost of software systems, "the true power of CASE tools only becomes apparent when they are all able to work together as a tool set" [6, 12]. The development of Software Engineering Environments (SEEs) has been the focal point of much research in the area of Software Engineering. A SEE can be defined as a collection of computer-based facilities to support the activities of programmers, software engineers, system designers, project managers etc. to achieve higher productivity and higher product quality [12]. A SEE is more than just a collection of tools in that it supports information passing between tools [13], and so offers to enlarge the choice for software developers of which tool sets (supporting methods and languages) to use in a given organisation or for a given development. It is recognised that the availability of such integrated environments is crucial for improving the productivity of the software industry. Integration is usually considered under three categories, presentation, control and data integration. Presentation integration is the "provision of a common user interface for the tools in an environment" [6]. Control integration is "the capacity to request operations from other tools in the system" while data integration is "the sharing and manipulation of information on which the various tools perform operations to satisfy requests"[43]. Most of the existing SEEs are based on at least two fundamental concepts of integration: control and data integration. It is these aspects of integration that concerns this thesis.

PCTE, Portable Common Tool Environment, provides standard services to support integration and portability of a SEE. PCTE is to a large extent object oriented, but the objects which it defines are data objects and do not have behaviour. These data objects however do provide data integration, information sharing for the tools, particularly suited to CASE tools. However due to this lack of behaviour, the amount of control integration or tool co-ordination within a SEE based on PCTE is limited. It is to address this limited control integration that we wish to integrate PCTE with Object Management Architecture (OMA), in particular the Common Object Broker Architecture, CORBA (the name given to the architecture of the Object Request Broker, ORB, component of OMA) the communications heart of the OMA specifications. Because the OMA specifications are specifically designed to promote development of integrated distributed systems using the object oriented paradigm, it makes sense to introduce OMA as an integration technology for PCTE, allowing PCTE to reap the benefits of being truly object oriented and increasing control integration between PCTE tools.

The integration could be beneficial from the OMA point of view, since the PCTE repository or object base could also be used as a persistent store for OMA objects. The Object Services component of the OMA which provides basic operations for the logical modelling and physical storage of objects, has not yet been fully specified and no implementations for it are available. Because the emphasis was on data integration within the PCTE specification, it, of necessity, developed a semantically rich data modelling mechanism, the Object Management System. Thus the Object Services component of the OMA would benefit from an integration of PCTE into OMA. These benefits all suggest that their convergence into a single standard is inevitable [5] and very attractive.

A considerable effort is being made by Object Management Group's PCTE SIG (Special Interest Group) to converge the PCTE and OMA specifications, see Section 1.2. However the purpose of this thesis is to provide an integration strategy for the

current PCTE and OMA's CORBA specifications, to their mutual advantage, while waiting for their convergence into a single specification.

The initial integration approach taken during this research was that of a direct language mapping between PCTE's Data Definition Language and CORBA's Interface Definition Language- such a mapping was sought because it would be a direct integration without altering the specification of either standard. However this approach proved unfeasible, for reasons discussed later in Chapter 5, and another integration strategy was sought, the same criterion being applied (no alteration to either the PCTE or CORBA specifications).

1.2 PCTE and OMA Convergence

In 1994 the Object Management Group formed the Portable Common Tool Environment Special Interest Group (PCTE SIG). The mission of the PCTE SIG is to provide support and requirements to OMG task forces for the convergence and interoperability of OMG's OMA and PCTE, specifically fostering PCTE compliance with OMA; to identify requirements for, and foster convergence of, interoperable CASE environments and fine grain repository tools for the evolution of PCTE [42].

The PCTE SIG now works with users, vendors, academia, government and provide technical liaison staff to work with relevant consortia and accredited standards organisations, to assure consistent requirements for the evolution of PCTE to OMA compliance, object orientation and fine granularity [42]. Substantial work towards the merging of PCTE and OMA has already been achieved. Work is currently in progress by OMG PCTE SIG on proposals for the object oriented extensions to the PCTE Standard (ISO/IEC -13719) which incorporate CORBA [49].

1.3 Background

Before we go any further, let us examine the importance of computer industry standards such as OMA and PCTE. Standards can have a profound impact on the way companies conduct their business. Consider the benefits already gained from the standardisation of such common languages as COBOL, C and SQL which enable the development of applications that are portable across heterogeneous platforms. When a standard is adopted and accepted, the direction of the industries that fostered the standard can be shaped for the better [40]. Take for example the popularity of Relational Data Base Management Systems (RDBMS) as opposed to Object Oriented Data Base Management Systems (OODBMS). Much of the success of RDBMSs comes from the standardisation that they offer, along with the simplicity and usability of the model. The acceptance of SQL standard allows a high degree of portability and interoperability between systems, simplifies learning new RDBMSs and represents a wide endorsement of the relational approach [41].

The software development industry is standing at a critical juncture where standards for the use of separate control and data integration strategies are beginning to emerge just as the benefits of the powerful new technological wave of software composition technology are becoming clear. Software composition is an approach to the creation of software by *composing* existing and new elements to form larger structures, writing a minimum amount of algorithmic code to do so. Composition technologies significantly reduce the effort required to build large software systems. For example, the developer of a chip-design package that integrates logical design, physical packaging and timing simulation does so by separately constructing the logical design component, the physical packaging component and the timing simulation component and then composing these into a complete tool [44]. Object technology greatly lends itself to the production of integrated software systems and software composition. This is why the members of OMG believed that the object-oriented approach to software construction best supported their goals of "developing and using integrated software

systems"[29]. While not necessarily promoting faster programming, object technology allows you to construct more with less code, partly due to the naturalness of the approach and also to its rigorous requirement for interface specification.

Tool composition refers to the software composition of tools, and in this thesis, CASE tools in particular. In order to enable such compositions, a tool integration model must permit the composer to chose a binding that is either high performance with tight coupling or lower performance with looser coupling. The difference is called the granularity of the composition. In general, small elements (fine-grained) require more frequent interaction and a consequent tighter coupling [44]. Platforms like PCTE are generally called coarse-grained because the intrinsic modelling and interpretative overheads and the implications of security and locking on an object-by-object basis limits its potential performance to coarse-grained interaction i.e. less frequent interaction and looser coupling.

The way in which standards relate to and are compatible with each other is also of importance to their success and acceptance. This thesis concentrates on the drawing together, in terms of an interim integration strategy, of two complementary standards within the computer industry namely OMG's OMA and ECMA/ISO PCTE in order provide a mutually beneficial integration. They are complementary in the fact that PCTE relies heavily on the data integration provided by its data modelling mechanism as a means of allowing tools to share common data, but an even tighter integration of tools- tool co-ordination is desirable, which can be provided by OMA's CORBA.

1.3.1 OMG & CORBA

In 1989 the Object Management Group (OMG) was established to simplify and reduce costs of software design and development and encourage the use of the object-oriented paradigm. To achieve this end OMG set down guidelines and object management specifications for a common framework, Object Management Architecture, of which CORBA (Common Object Request Broker Architecture) is the specification for the communications component of OMA. OMA is a set of specifications designed to support applications that are collections of interoperating, co-operating distributed objects. Following industry's adoption of these specifications, they will be instrumental in the standardisation of the object technology and make it "possible to develop heterogeneous applications environment across all major hardware and operating systems"[42]. Since OMG is an open consortium, with over 500 members world wide, the specifications are set by industry itself, thus ensuring their relevance and the dedication of the computer industry to their acceptance. Chapter 3 contains a description of OMG's CORBA and its place within the broader Object Management Architecture.

1.3.2 PCTE

Of the considerable variety of CASE tools now available on the market, very few of them can be easily integrated for example, coding, design and testing tools which support only isolated aspects of the software process. Tools which can only work in isolation are useful to a point, but do not fulfil the potential of a SEE. A SEE can be described as in [6] as consisting of four layers : Platform, Public Tool Interface, Framework and Environment (See Section 2.2). One approach to the efficient integration of software engineering tool sets is to factor out those common features required by most tools for information management and interaction with the tool users. It is therefore an efficient way forward to define a domain in which these common

needs are satisfied, whilst leaving the tools themselves to carry out the specific tasks and offer their specific facilities. Out of this realisation has come the concept of the Public Tool Interface. The Public Tool Interface (PTI) is the layer which provides standard services to support integration and portability [12].

Portable Common Tool Environment (PCTE) is an example of one such PTI for an open repository. It defines a set of public and standard services designed to support portable and well-integrated CASE tools. A repository is a place for storing all the information that is required in a software engineering environment, for example tools, software products and documents. The Object Management System (OMS) within PCTE provides the functions used to access the repository. PCTE provides a public schema mechanism that allows independently sourced tools to access and manipulate information in the repository [6]. The repository approach to data integration has been the main focus of the IRDS (Information Resource Dictionary Systems) and PCTE efforts. The IRDS advocates have strived for nearly a decade towards the illusive goal of acceptance within the standards community [40], compared with PCTE which could be viewed as the leading repository standard, having gained ISO standardisation in 1994. Control Integration for CASE tools has been pursued via a standardised message (structure and semantics) as promoted by the CASE Interoperability Alliance (CIA) and CASE Communiqué industry groups and recently X3H6 [5]. Chapter 2 describes the PCTE standard in detail.

1.4 Technical Issues

This thesis explores the very real alternative of using the distributed object approach (i.e. CORBA) to provide control and data integration for the general problem of integrating CASE tools in a SEE, using a single paradigm (Since CORBA objects have both state and behaviour as opposed to PCTE objects which have only state).

PCTE and OMA specifications are both concerned with the integration of distributed applications. OMA applications are collections of interoperating co-operating distributed objects (data and methods) ranging from large to fine-grain objects. OMA is suitable for a wide range of domains including CASE [5], which encompasses the focus of PCTE, the area of data integration for distributed CASE environments.

The major technical issue that is solved by layering CORBA on top of PCTE is concerned with the fact that CORBA implicitly defines "distributed and secure execution and interprocess communication services" [5]. Thus the introduction of CORBA as one of the integration technologies of a PCTE SEE is beneficial from the perspective of the tool integrator and the framework builder, in that CORBA supports tool composition, and therefore will provide greater control integration between PCTE tools. It will also allow the definition of tool interfaces so that they can make their services available to the rest of the environment, while hiding the implementation details [43].

The support of tool composition, as discussed in Section 1.3, is an important part of tool integration for CASE. The ambitious goal of tool composition is to allow construction by assembly of separate pieces of systems that have the usability, and almost the performance, of hand-crafted monolithic systems. The advantages of composition are ease of construction, reuse of components and ease of extensions and maintenance. The challenge is to maintain usability and performance. Fine grained composition in which components can interact tightly and frequently and can share small granules of data is essential to meeting this challenge. PCTE and CORBA both contribute in largely complementary ways to supporting composition, and so their integration into a single platform is attractive [44].

From the OMA/CORBA point of view the PCTE repository could be used as a persistent store for OMA objects, thus permitting the state as well as the methods of OMA objects to be stored. The persistent storage of objects is a part of the Object Services component of the OMA which has not been fully specified yet (see Section 3.3.1).

The implementation of PCTE used during this research was GIE Emeraude PCTE Environment V12 and the implementation of CORBA used was IONA Technology's ORBIX Version 1.1.

1.5 Overview

The remainder of this thesis is structured as follows. Chapter 2 contains details of the PCTE standard, including its Object Management System and Data Definition language. It also contains a description of the typical structure of SEEs based on PCTE and a discussion on the industrial relevance of the PCTE standard and its currently available implementations. Chapter 3 contains a description of OMG's CORBA, its place within the broader Object Management Architecture, its Interface Definition Language (IDL), the role of OMA within the computer industry and the availability of CORBA implementations and CORBA-compliant products. Chapter 4 examines the complementary relationship that exists between PCTE and OMA. It discusses the approaches to integration researched in this thesis while looking for an interim solution to the integration of PCTE and OMG CORBA, and the benefits that would hope to be achieved by such an interim integration. Chapter 4 also contains details of related work in the area of the integration of PCTE and OMA/CORBA and discusses where the research contained in this thesis fits in relation to this work.

As will be discussed in Chapter 4 the most obvious choice for an interim integration strategy is that of a direct language mapping between DDL and IDL (and vice versa). Much of the work of this research was to prove that such a mapping will not be possible until the specification for DDL (in particular) has been extended. The fact that this thesis proves that a mapping between DDL and IDL is not currently possible is a valuable contribution in itself. Chapter 5 outlines the nature of a DDL to IDL mapping given the current specifications and discusses why such a mapping is proven to be not viable as an interim strategy for PCTE and CORBA. Chapter 5 also contains a description of the extension DDL would require for compatibility with IDL, in order to make such a strategy viable for future integration.

Chapter 6 describes the definition of IDL interfaces for PCTE tools as an alternate approach to the interim integration of the two standards, and discusses the benefits of this strategy. Chapter 6 contains a demonstration of how the definition of IDL interfaces for PCTE tools may be used to increase control integration between tools in a PCTE repository (in the demonstration Emeraude PCTE V12), to support tool composition for PCTE tools, and to enhance PCTE objects to full object orientation. Chapter 7 concludes by providing a summary, evaluating the usefulness of the integration strategies researched in this thesis, and how future work can build upon this research.

CHAPTER 2 PCTE

This chapter introduces the history and concepts behind PCTE, particular attention is being given to areas of the standard which are deemed important in context of this thesis. Section 2.2 contains an introduction to the architecture of PCTE-based Software Engineering Environments (SEEs). The PCTE repository or object base and OMS (Object Management System) are described in Section 2.3. Section 2.4 describes the Data Definition Language (DDL) which is PCTE's data modelling and integration mechanism. PCTE processes are introduced in Section 2.5, and a description is given of the way in which they are used as mechanisms by which the PCTE repository is interrogated and modified. Section 2.6 describes PCTE activities and how they are used to ensure the consistency of the PCTE repository. Section 2.7 describes the implementations of PCTE available on the market and discusses the industrial relevance of the PCTE standard.

2.1 PCTE - The Standard

As already mentioned in the introduction, a variety of CASE (Computer-Aided Software Engineering) tools now exist, the function of which is to automate aspects of the construction of software itself. Until the advent of integration standards such as PCTE, software was developed predominantly on large centralised computer systems using a collection of tools, with minimal data integration at file level. These tools supported isolated aspects of the software process, and bore little or no relationship to one another- e.g. coding, design, testing tools respectively supporting only the coding, design and testing stages of process development. Although such isolated CASE tools did have the potential to reduce the production cost of software systems, the full potential of CASE tools is only apparent when they are able to co-operate together as part of a tool set. Therefore the basis of any CASE environment must be a flexible framework which offers a "cost-effective tool integration mechanism, encourages portable tools, and facilitates the exchange of development information"[25].

Tool integration is defined as a property of a tool's relationship with other elements of the environment, chiefly other tools, the platform and a process [26]. The complexity and interrelation of CASE components require an environment supported by comprehensive standards that allow a range of tools and techniques to work properly together[27]. The highest degree of CASE integration is achieved through the use of a standard model for tools. Such a standard defines what mechanism a Software Engineering Environment (SEE) or tool developer has to use for tool communication, the representation of the user interface, and the data model within the Repository. PCTE, the Portable Common Tool Environment, is such a standard for "a public tool interface for an open repository"[6]. A Public Tool Interface (PTI) is defined as a set of program libraries that grants access to facilities and services needed by tool writers and environment builders[28]. In order to support a high degree of integration as well as portability of CASE tools, PCTE defines a set of public and standard services and uses the PCTE repository to store the necessary information associated with a Software Engineering Environment. The information stored in the repository may include documents, source and compiled code for the products under development, as well as the CASE tools themselves.

Before we examine the role of PCTE within a Software Engineering Environment, let us turn briefly to the origins of the PCTE standard. PCTE was initiated in 1983 by the European Strategic Programme for Research and Development in Information Technology (ESPRIT) as a project called "A Basis for a Portable Common Tool Environment". That project, partially funded by the Commission of the European Communities, produced a specification for a tool interface, an initial implementation, and some tools on that implementation. The objective of this interface was to allow the building of SEEs and promote their implementation on different hardware and operating systems. Following the acceptance of the first edition as an ECMA standard in December 1991, review by international experts has led to the production of a second edition taking into account review comments relating to this standard. This edition was accepted as an ECMA Standard by the General Assembly of June 1993 [1]. In 1994 PCTE became an international standard, as ISO/IEC 13719.

2.2 Architecture of PCTE-based SEEs

The architecture of PCTE-based Software Engineering Environments are described in this section according to the following four layers : platform, the public tool interface, framework and environment. Such a description of a SEE is based on concepts originally described in the PCIS Technical Study paper [24].

The platform consists of the hardware of a machine and the operating system needed to use it. Essential to Software Engineering Environments is *portability*, the capability of using the same software on different platforms. Portability is important because the users of an environment want to make use of their favourite tools regardless of the platform on which the environment is based, this being part of the overall current trend towards "plug 'n' play" tools (i.e. tools which can be slotted into an environment with ease regardless of their vendor). In the classic "Non-Open" model, interfaces between tools have to be modified each time a new tool is introduced. This prevents the plug and unplug replacement of tools and means the user is restricted to a particular tool vendor. The portability of software can be reduced due to a number of factors including differences between machines (software developed for a given machine architecture utilises specific features of that architecture making it unportable for other architecture types) and differences between operating systems (programs which contain operating system calls may not be portable). The PCTE Public Tool Interface is designed to increase the portability of tools.

The Public Tool Interface (PTI) hides the platform and provides a uniform base on which software can be developed [6]. PCTE is an example of such an interface for an open repository, as it is designed to shield applications from the variances among differing platforms. The PTI defines a set of interfaces, these usually being implemented as a set of operations on a given platform. In the case of a PCTE-based SEE these operations are obviously PCTE operations (See Tables 2.1 and 2.2 for example). The Public Tool Interface is a non proprietary, public and widely available standard to which all tools in an environment should conform. Tools which are in exact conformance with the PTI are portable to all platforms on which the PTI is implemented.

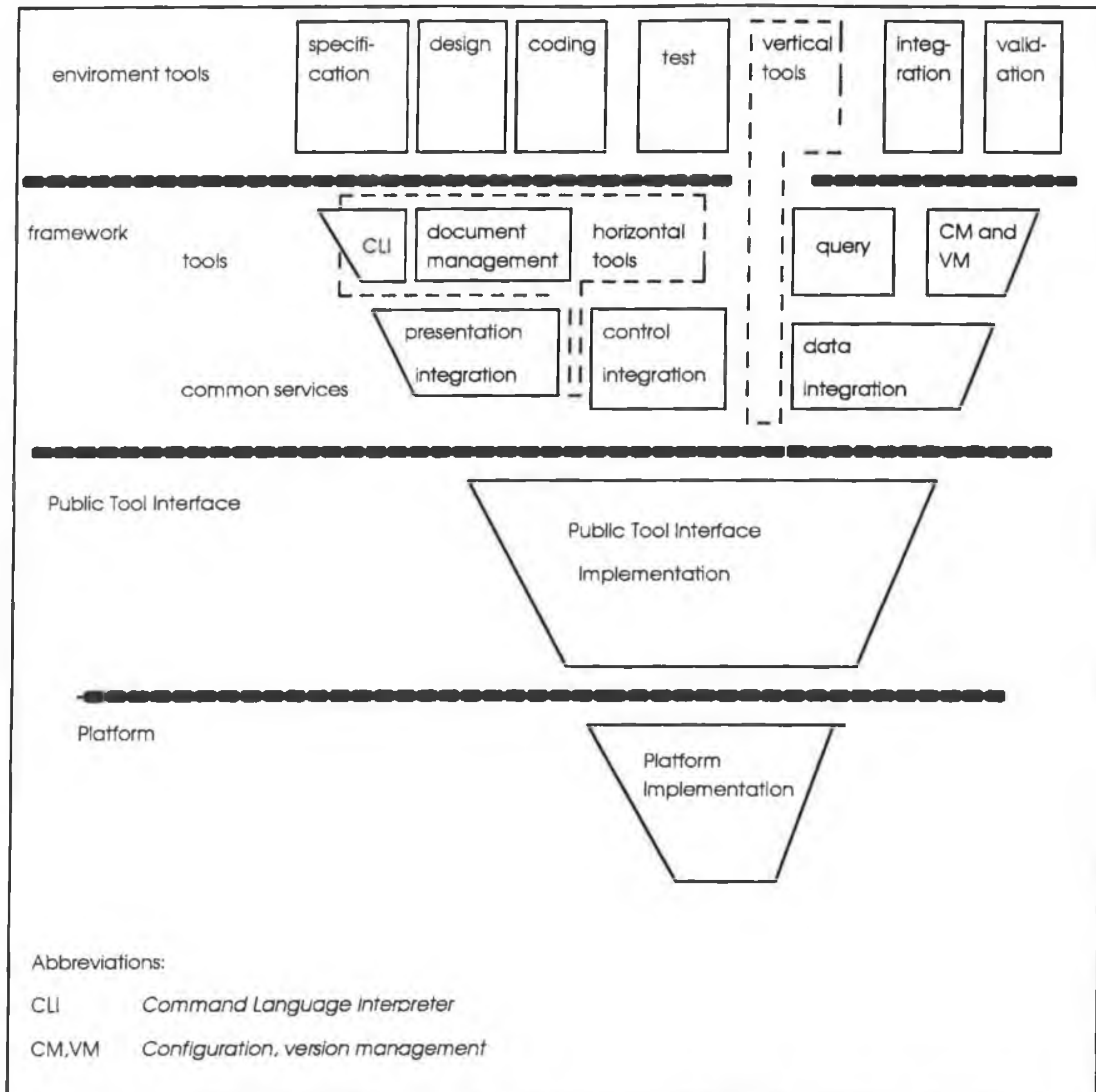

Figure 2.1 *An architecture for a Software Engineering Environment*

The framework incorporates the Public Tool Interface along with general purpose horizontal services and tools whose functionality is generic to all stages of software development and maintenance e.g. repository browser and querying tools, configuration management tools, communication and documentation support tools. One such PCTE framework, Émeraude V12 is described in diagram 2.2. Émeraude V12 includes PCTE libraries implementing the PCTE interface and builds on these common services for the management of metadata, data query, version and configuration management. It provides a number of horizontal tools, including some basic PCTE tools (e.g. to create objects and links, set attributes) and encapsulation of UNIX tools[6].

It is the framework layer which provides support for data, control and presentation integration within the complete environment, and it is this integration which sets apart the SEE from a set of independent tools executing on an operating system.

A joint development by ECMA and NIST addressing the area of SEEs, particularly the services that are expected to be useful in an environment framework, has provided a reference model [21]. The reference model puts the framework services into a number of groups, which are commonly represented by a diagram often referred to as the "toaster model" see [6, 20]. Figure 2.3 shows how a PCTE-based SEE would be represented using the this model.

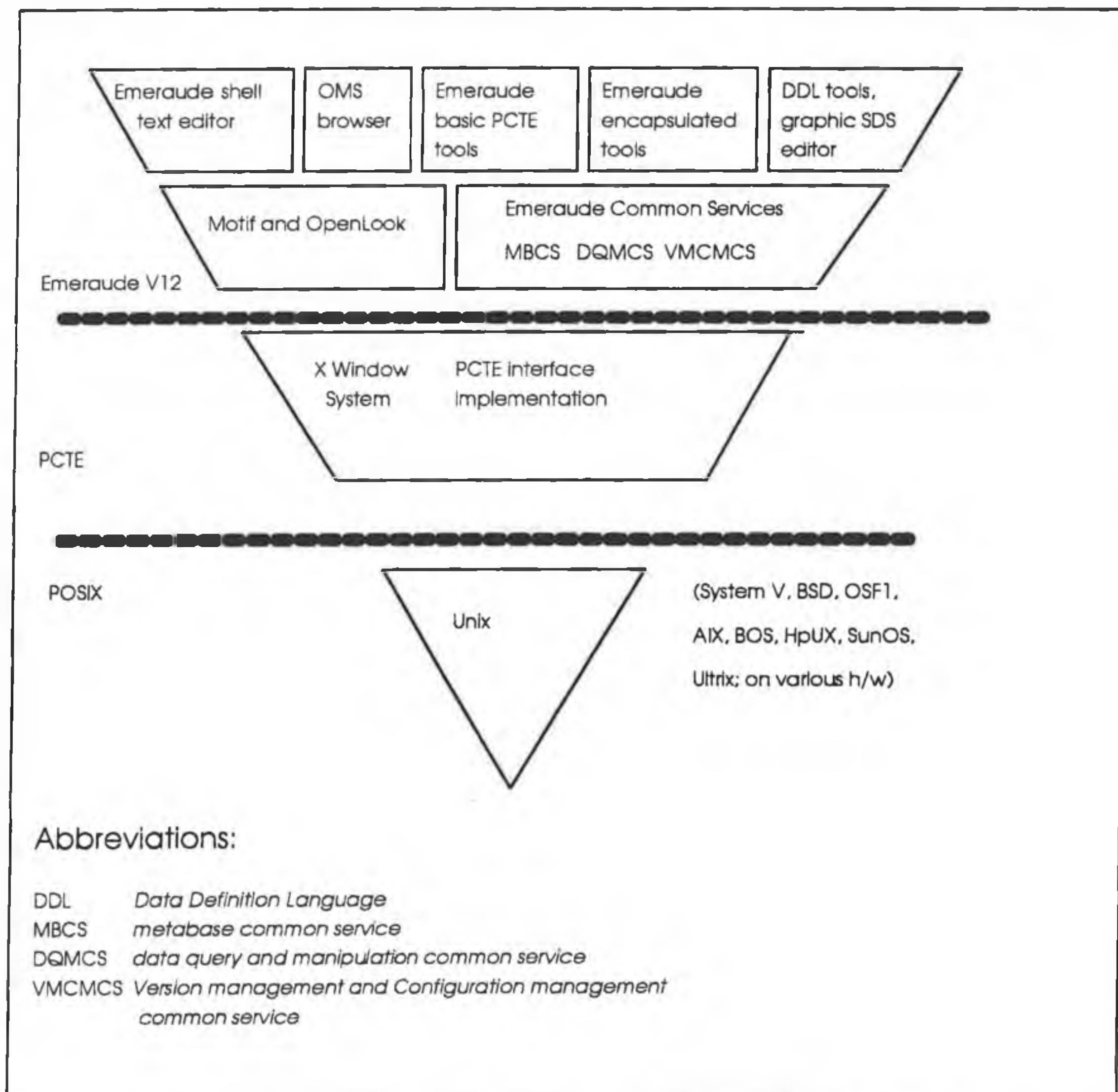

Figure 2.2 The Émeraude V12 Framework

The Environment includes specialised vertical tools to support the life cycle, along with the other layers outlined previously. These tools extend the basic capabilities of the framework to application-specific domains (e.g. CASE, CAD (Computer Aided Design)). PCTE addresses the CASE domain in particular. Each of the vertical tools are designed for use in a particular phase of the software development process. These phases may occur sequentially, as described in [3], or as a cascade, the so-called waterfall processes, as described in [4]. In an extendible SEE, its set of horizontal and vertical tools may evolve due to the altered requirements of an organisation or the availability of newer and better tools. To enforce a higher level of integration, the environment may enforce the use of common services for all tools, this however being at the cost of restricted usability of imported and foreign tools.

Emeraude Components

Object Base/Repository
Object Management Services (OMS)
Emeraude VCM
Emeraude PCTE Tools & Services
(inc. OMS browser,
Text Editor,
Schema Editor,
OMS Utilities)
Communication Services

Partially provided by Emeraude PCTE

User Interface Service
CASE Tools

Figure 2.3 PCTE-based SEE using the toaster model.

Having seen in this section the role of PCTE within a SEE, sections 2.3 and 2.4 examine how this role is fulfilled by PCTE OMS (Object Management System) and Data Definition Language.

2.3 PCTE's Object Management System

Central to the process of constructing and integrating portable tools by PCTE is the provision of the object base and a set of functions to manipulate the various objects in it. "The object base is the repository of data used by the tools of a PCTE installation, and the Object Management System or OMS of PCTE provides the functions used to access the object base" [1]. The OMS can be seen as an evolution from a traditional File Management System (e.g. the hierarchic structure of UNIX) to a structure that can be adapted to the needs of different environments [13, 14]. The OMS is designed to enable the transparent distribution of the object base over a local area network.

PCTE makes use of database system technology and a complex object model as well as semantic data model theories (see [18, 23]) to overcome the shortfalls of applying traditional database systems to a SEE repository [12, 16]. The object base, a relational database, is the repository of persistent information that is employed by software tools [17]. The object base or repository is required to store and manage very complex data and relationships across the whole software life cycle- not only finished products of the software process (e.g. designs, functional specifications, alpha, beta and fully tested versions of code, fault reports, change requests) but also the intermediary and supporting data that accumulates along the way (e.g. project history, test results, memos and reports) [6]. The basic OMS model is derived from the Entity Relationship data model and defines objects and links as being the basic items of a PCTE object base [1, 12, 15]. The object base can be viewed as a directed graph, in which the objects are nodes, and the links are arcs of the graph. This network of objects, connected by links, allows a large number of complex relationships to be modelled in an intuitive way, see figure 2.4 for example.

2.3.1 PCTE Objects

PCTE objects are entities which can be designated, and can optionally have :

- contents — a storage of data representing the traditional file concept.
- attributes — primitive values representing the specific properties of an object which can be named individually.
- links — representations of association between objects. Links may have attributes, which may be used to describe properties of the associations or as keys to distinguish between links of the same type of object.

Designation of links is the basis for the designation of objects: the principal means for accessing objects in most OMS operations is to navigate the object base by traversing a sequence of links [1].

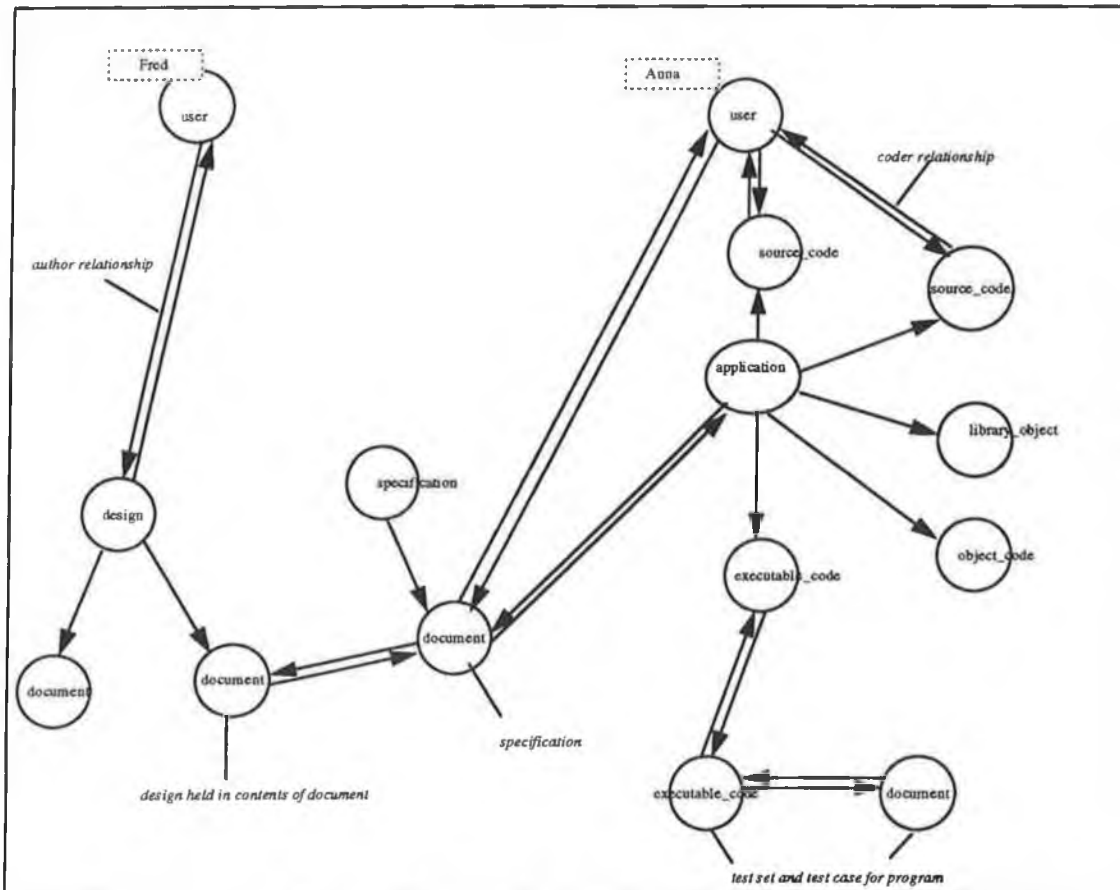

figure 2.4 Some example objects in a PCTE repository

2.3.2 Links

Links represent the relationship between PCTE objects, which (the relationships) can be uni- or bi-directional. However each link has only a single direction. Bi-directional relationships are described using reverse links, whereby two links with opposite direction join the objects, each of which is called the reverse of the other. As stated above, links (similar to objects) may have attributes. Object attributes record a specific piece of information about an object, whereas link attributes record something about the relationship between two objects.

2.3.3 PCTE Operations

The OMS supplies link and object operations for the basic manipulations such as creation, deletion and the setting of a value for an attribute. In addition it has a set of operations that give access to contents (files, pipes, devices) of an object. However, the OMS cannot decipher these contents, its meaning being left to the software tool[2, 12].

Layered on top of the OMS are various services covering execution (process management), interprocess communication, activities (transaction management), distribution, discretionary and mandatory security, notification, accounting and object contents operations. Many of these services are modelled as objects in the OMS. For example, processes are modelled as objects and the properties of a process are expressed in terms of object attributes and links. This approach allows uniform information access [5].

In the PCTE specifications [1], the abstract operations and their bindings are categorised by function in a number of separate but often related clauses [6]. The main functional areas are summarised below to indicate the scope of the PCTE interface:

Object management & schema management

operations to define and manage, in a general way, all the instances and typing information in an object base (objects, links and attributes).

Object contents

operations to manipulate the data stored in the contents of objects.

Process execution, message queues & notification

operations to manage the execution of programs and their communication

Concurrency and integrity control

operations to prevent loss of data integrity by locking and transactions.

Discretionary security, mandatory security & auditing

operations to enforce security policy and to audit object base accesses.

Volumes, devices, archives and replication

operations to manage the distribution of data, to represent physical devices and to make archive copies of objects.

Network connection
operations to manage the configuration of workstations.

Accounting
operations to monitor resource use.

2.3.4 PCTE Types

An essential principle of the OMS is that of schema. The schema is a means of integrating tools around commonly accessed data structures [13]. Rarely is it convenient or useful to model an entire environment as a single system. It is important that the SEE framework supports the break down of the object base into sets of data that model different aspects of the environment and development process. In PCTE, this functionality is provided using the working schema mechanism. Each working

schema represents a specific view (e.g. tool's, user's or project's view) of the object base. Central to the understanding of schema is the PCTE notion of *types*.

A type "captures the essential characteristics of like entities from the same domain, abstracting their common properties"[6]. For example all sources contain text, all programs contain executable binary code. We can identify many classes of objects that share common characteristics, for example, the relationships that a class of objects may have with other classes of objects, or the kind of attributes that are relevant. PCTE represents these characteristics with types, each of which is a data definition for a particular class of object. A type may be defined for each class of similar objects within the repository- take for example a type defined for specification documents, design documents or source code objects. ADA *source*, C *source* and FORTRAN *source* are each a specialisation of the general *source* type with their own particular characteristics in addition to the general ones.

When any object base entity is created, it assumes the characteristics defined by the type; these characteristics can govern, for instance, the number of links of a type that can leave an object or the reverse link of a link. The type definitions making up the overall OMS schemas are organised into a collection of small sets of definitions called Schema Definition Sets (SDSs) [13]. When part of the OMS schema must be visible for a particular purpose, a SDS is used to represent this partial view of the OMS. An SDS defines both the visible characteristics of the basic entities (objects, links, attributes) and how they can be related to each other: a link of this type can have this object type as its origin, and that type as its destination; this attribute type is applied to this object or link type. For example a project scheduling tool might use a set of related types representing projects, milestones, start dates and estimated duration of project tasks. Each PCTE process (see Section 2.5) has a working schema, a mechanism by which sub-schemas represented by SDSs can be used by processes, and therefore the means by which running tools are presented with the schema representing their data models. A working schema is a linear collection of SDSs and all environment referencing is done through the working schema [12]. A tool's view of the repository is through its working schema. Thus only the types belonging to the SDSs contained in its working schema's list of SDSs will be visible to any tool.

This typing mechanism is fundamental to the data integration of tools within PCTE, by allowing tools from different vendors to have their own names for common classes of objects. The data model of a tool can be compared with those of other tools and, by identifying the common data entities, the models can be integrated with each other and implemented as common types. In the PCTE typing model, when new types are defined they must be integrated with the existing types of the environment (every PCTE based environment pre-defines the same four SDSs **system**, **metasds**, **security** and **accounting**, these being necessary to support the operation of the interface). The separation of a tool's data model from its code, as provided by SDSs, greatly facilitates data integration.

2.4 PCTE's Data Definition Language

The PCTE Data Definition Language (DDL) is a formal notation for defining types [6], and a schema definition formalism [14] for SDSs. It is a convenient textual (as opposed to graphical or functional) notation. The syntax of DDL can be found in Appendix B of this thesis. This section gives a brief introduction to the semantics of the language. To illustrate these semantics the c_prog SDS will be used, see Appendix C for its complete DDL listing. DDL is normally divided into SDS sections. SDS sections group together type definitions, each DDL type being declared within a SDS section.

The DDL declaration of a type includes both the type and the *type-in-SDS* properties for the current SDS. As stated in the previous section each newly defined type must be integrated with the existing types of the environment (the pre-defined types within the **system**, **metasds**, **security** and **accounting** SDSs). The *type-in-SDS* properties exist to represent the properties of object, link, attribute and enumeration item types as they are used within a particular SDS. Remember that each SDS is a view of part of the whole OMS schema; a part must be visible for a particular purpose. For example the c_prog SDS is used to model an environment for the C programming language, that is an environment which can describe, for example, the relationship between C source code files, object code, libraries and header files. Therefore the c_prog SDS defines only the type objects which would be of interest while programming in C, and leaves the remainder of the object base hidden (for example any FORTRAN source files would be hidden). The SDS definition for c_prog first imports types which are

intrinsic, and then applies specific properties to them which are necessary to describe the C programming environment.

Some properties of PCTE types are intrinsic, for example, the kind of contents an object of a particular type can have, or the value type and initial value that the attribute type can have. They are intrinsic in the sense that they are assigned when the type is created, are unchangeable, and are therefore the same in all SDSs in which the type subsequently appears. Even if an SDS applies other properties to the type (these applied types are the *type-in-SDSs*), the intrinsic properties cannot be changed. Applied properties include the attributes of an object or link type and set of link types leaving the an object type. We will take as an example the definition of the *program* type from the c_prog SDS:

import *pact_software* **as** *program;*

defines the pre-defined type pact_software (defined in the pact_sds) for use within the c_prog SDS; within this SDS the type will be renamed as *program.*

extend *program*
with
attribute *version*;
edition;
system;
system_release;
target;
variant;
link *deliverable*;
sub;
inc;
build;
tests;
exec;
subprog;
a;
out;
end *program*;

Program is then extended to define properties which will apply to this type (*program*'s type-in-SDS) within the c_prog SDS. As the DDL listing above illustrates, these applied properties are to include new attributes, namely: ***version***, ***edition***, *system*, *system_release*, ***target*** and *variant* which will apply to ***program*** within the c_prog SDS. Furthermore, ***program*** is restricted so that only links of the following type are allowed to leave a ***program*** object: ***deliverable***, ***sub***, ***inc***, ***build***, ***tests***, ***exec***, ***subprog***, ***a*** and *out*.

Data schemas are explicitly represented in PCTE by data instances referred to as the metabase. The metabase deliberately distinguishes the intrinsic properties of a type from those applied to it within the context of a specific SDS. It contains a set of objects (so called types-in-SDS) which represents the specific properties that a type holds within a given SDS. The distinction between intrinsic and applied properties is important to the understanding of schema installation and evolution strategies. SDSs are primarily sets of types with applied properties and can be managed as such [23].

Within an SDS section, the types are defined in a relatively free order and flexible way. Because of this flexibility there are often two or more equivalent constructions to declare the same types [6]. Types can be defined in either single or compound declarations with related types. All DDL types (Object, link, attribute and enumeration item types) can be imported from other SDSs or declared within a SDS; object types can be declared within a SDS as descendants of existing or previously imported types, the method by which they are imported is the DDL Type Importation declaration. In the following Sections 2.4.1 -2.4.6 the SDS section in which a clause occurs, and the SDS to which it contributes, are called the ***current*** SDS section and SDS respectively. The examples used in these sections illustrate the language constructs provided by DDL for defining types.

2.4.1 Type Importation Declaration

A type importation declaration defines a type in SDS in the current SDS. This type in SDS is a copy of the type in the SDS from which it is being imported. For example the following DDL type importation declaration, taken from the c_prog SDS:

import object type *pact-env* **as** *env* **(usage navigate; export protected) ;**

imports the object type *env* from the *pact* SDS; this imported object type will also be known as *env* within the current SDS. Navigate and protected are the usage and export type modes which govern how the type may be used within the SDS (PROTECTED, READ, WRITE, DELETE, CREATE, NAVIGATE) or if it may in turn be exported from the current SDS.

2.4.2 Object Type Declaration

An object type declaration always specifies the parent type(s), and may also name or declare the applied attribute types and out-going link types. Only the basic pre-defined child types of object can include a contents type specification. The following example is taken from the pre-defined **system** SDS, and is the DDL definition of the file type object:

file:**child type of** *object* **with**
contents file;
attribute
        contents_size: **(read) natural;**
        positioning;
end *file*;

This DDL object type declaration specifies the type *object* to be the parent of the *file* type, to have contents (**file**) and attributes called *contents_size* and *positioning*. In turn the *file* object type may be the parent type of another object.

2.4.3 Link Type Declaration

There are several ways to specify a link type and its application in DDL. A link type can be named or declared, and applied to its origin and destination types, within an object type declaration; this is the case in the *program* object declaration given earlier

in this section. Alternatively a link type and its destination type can be declared together. The definition of the link tests in the c_prog SDS illustrates this type of link declaration:

tests: **composition link to** *testset*;

This DDL definition states the object type *testset* to be the destination of links of the type *tests*.

2.4.4 Link Type Extension

An existing or imported link type can be applied to new destination types in a link type extension. Take for example the following DDL link type extension declaration:

extend link type *tool* **to** *sctx*
with
attribute
user : **string** ;
end tool ;

Here the link type tool is given a further outgoing destination type and attribute type, sctx and user, respectively.

Also an existing or imported link type can be applied to new origin types in an object type extension, as shown in the next section.

2.4.5 Object Type Extension

An object type extension extends the object type-in-SDS in the current SDS, by adding further outgoing link types, attribute types, component object types. For instance in the object type extension of testset within the c_prog SDS (shown below), *tst* and *theme* are declared to be further outgoing links for the object type *testset*.

extend *testset*

with
attribute *nature*;
link *tst*;
theme;
end *testset*

2.4.6 Attribute Type Declaration

There are also several ways to specify an attribute type and its application in DDL. An attribute type can be declared independently, without any applications. Take for example the DDL attribute declaration of *version* within the c_prog SDS:

version : **integer := 1;**

An attribute type can be declared (if it has not already been declared elsewhere) and applied to an object or link type within an object or link type declaration or extension, as the *version* attribute was applied to the object type *program* shown earlier. An existing or attribute type can be applied to new object or link types within an object or link type declaration or extension [6]. Enumeration Item can either be defined independently or within an enumeration attribute type declaration.

As can be seen from the above sections, PCTE has an inheritance mechanism for object types. A child type inherits the properties of its parent type(s). In addition it may have properties of its own such as link types, attribute types, contents and components [1]. It is important to note however that, although PCTE supplies the concept of inheritance, it is not a truly object-oriented environment. The reason for this observation is that PCTE's DDL does not have a mechanism to attach methods to object types. Such a mechanism is expected from a truly object-oriented environment [12].

At this point, we have seen how portability and integration are supported by PCTE. The portability of PCTE tools to other platforms, where PCTE is implemented, is guaranteed by the conformance to the interface provided by the Object Management System. Integration for tools is provided through a data integration mechanism

available in the form of DDL definitions for a tool's working schema. We will now examine the nature of PCTE tools, which are the mechanism by which the repository is interrogated and modified.

2.5 PCTE processes

A PCTE process is the execution of a program, whether this is a software engineering tool or general tool, a target application, or one of the components of the PCTE implementation [6] (PCTE tool, or tool, may be used to mean PCTE process). Program code for these PCTE tools is stored in static context objects in the object base. A static context (short for static context of a program) is an executable or interpretable program in a static form that can be run by a process. An executable static context can be loaded directly and then executed. An interpretable static context is a program that can be run by a process but it first requires the running of another static context as an interpreter. A static context may be run either by a PCTE implementation or by a foreign system [1]. A foreign system can be a foreign development system, a target system running a real-time operating system, or even a PCTE workstation in another PCTE installation.

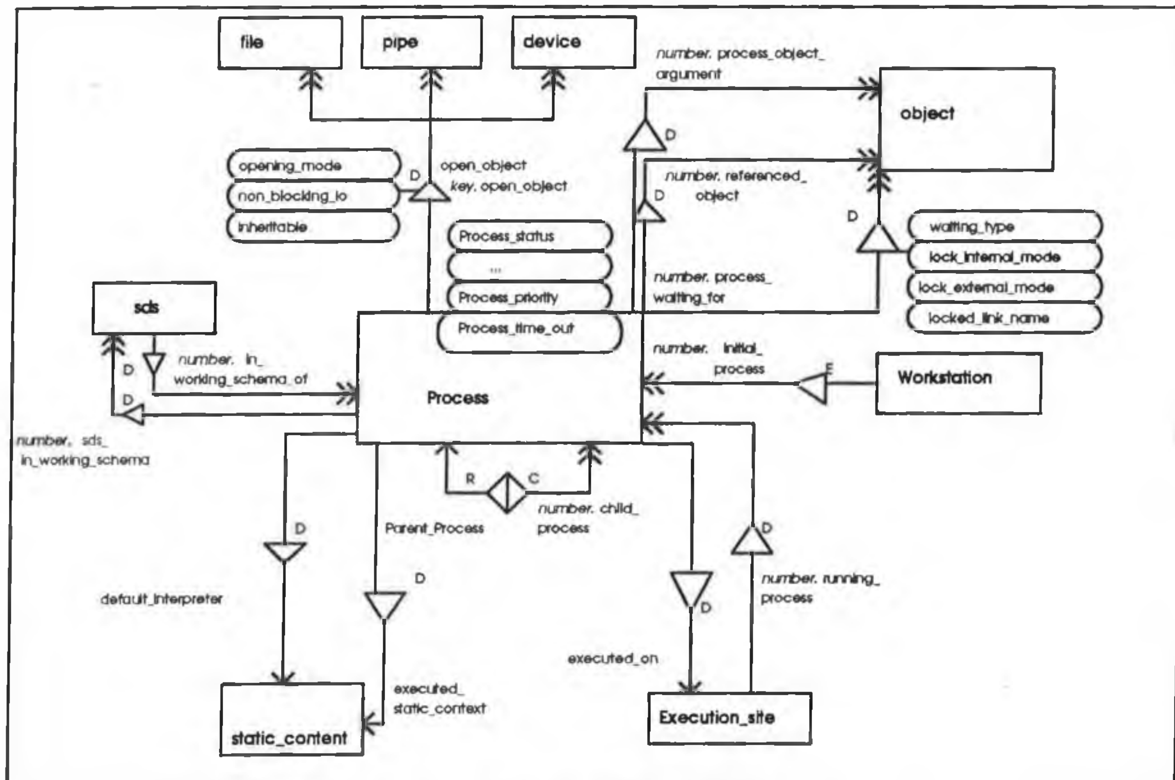

Figure 2.5 The Process Execution Schema

A PCTE process is a dynamic entity which has a temporary object base representation; an active process has a dynamic context which is the collection of all its properties; including its data and code images in memory, working schema, reference objects, named variables [6]. A PCTE process can be looked upon as a means of running a static context i.e. a dynamic context is a running static context. One property of the dynamic context of a process is its parent process, i.e. each process must have a parent process. The first process created to realise a PCTE environment on a given workstation is the PCTE initialisation process, and from this initial process a tree descends. A child process can be executed on a different site from its parent; therefore a process tree starting on one workstation often becomes distributed over several workstations of a PCTE-based environment.

Processes are created and started in two separate steps :

- When a process is created, so is the process object and many of its links. At this stage, the instances in the object base represent how the dynamic context will be initialised. It is possible for the new process, or another one, to change many of the properties to suit the static context that is to be run (e.g. defining the working schema with which the program is to work and setting referenced objects to designate the objects to be processed). For example a browsing tool may change the working schema and set new reference objects at the user's request. The processing of most of the PCTE operations depends on the dynamic context of the calling PCTE process (for example, security checks performed by an operation depend on the discretionary and mandatory context of the calling process; visibility checks are done on the basis of the current working schema of the calling process etc.) [22].

- When the process is started, additional links are created and attributes initialised. At this second stage, the instances in the object base represent the dynamic context itself.

The success or failure of a process is determined by the static context that it is running. Upon process termination, a program defined result and termination status becomes available from the object base to other related processes. The result indicates

successful exit or abnormal termination. The termination status indicates whether the object base instances will be deleted when the process terminates and that termination is acknowledged by its parent. Figure 2.5 shows the process execution schema. Schema diagrams are a frequently used graphical method of illustrating actual or example types and their relationships. The conventions used for schema diagrams are outlined in Appendix B of [6]. Table 2.1 lists some of the operations provided by PCTE for managing processes.

The tools in a SEE may be composed of several more rudimentary tools, each running its own process. The rudimentary communication of results from a child process to its parent is not sufficient for the co-operation and synchronisation that is needed between the components of a tool, which may quite possible be running in unrelated processes. PCTE also provides basic facilities for interprocess communication in such an event, such facilities amounting to the only control integration facilities provided by PCTE.

PCTE Process Operations	
PROCESS_CREATE	Creates a process ready to run a static context, as the child of the parent process.
PROCESS_START	Starts the execution of a created process.
PROCESS_SET_REFERENCED_OBJECT PROCESS_UNSET_REFERENCED_OBJECT	Sets a referenced object of a process to a specified object, and unsets a referenced object respectively.
PROCESS_SET_WORKING_SCHEMA	Sets the working schema of a process to a set of SDSs.
PROCESS_WAIT_FOR_CHILD	Makes the calling process wait for a child process.
PROCESS_SUSPEND	suspends a running or waiting process.
PROCESS_RESUME	Resumes a suspended process

PROCESS_INTERRUPT_OPERATION	Interrupts a process executing a PCTE operation.
PROCESS_TERMINATE	Terminates, and in some cases deletes, a process.

Table 2.1 Some of the PCTE Process Operations

2.5.1 Interprocess Communication

PCTE provides message queues and pipes as low level means for processes to communicate with each other. Message queues are objects which provide an independent place in an object base to store messages from a process, which can be accessed by other processes. The contents of message queue objects are a sequence of individual messages; the meaning of the message text is understood by the posting and receiving tools, which therefore must be designed to co-operate. Notification does allow limited co-ordinated use of tools which have been independently developed. A notifier is an association between a process and another object that allows the process to watch for access events on the object. Various kinds of access can be monitored- for example the modification of an object in any way, its removal to another volume or its deletion [6].

Message queues provide for a structured communication between processes. Sometimes a simple transfer of information, perhaps in large volumes, is what is required. A pipe is an object whose purpose is to have sequential data written to it and then read from it by processes [6]. However data can be read from a pipe only once, after which it is no longer accessible.

All accesses to, and modifications of, an object base are actually performed by PCTE processes. However for concurrency and integrity control purposes, access is managed in the context of activities.

2.6 PCTE Activities

PCTE is designed to ensure that the object base is never in an inconsistent state due to the failure or partial failure of an operation. An activity is a way to manage a set of processes that are performing actions related in some way. Concurrency and integrity control of the object base is managed using activities. Operations (one or more) are carried out within an activity using one or more PCTE processes. An activity can hold locks on the entities that it is using to protect them from access attempts by other activities. Transactions are the mechanism by which PCTE maintains consistency. Because PCTE supports the nesting of transactions, it is possible to build tools from existing tools or re-usable components without concern for individual error recovery actions [6]. Transactions are a special class of activity, which have the property of atomicity (either all their constituent operations are committed to the object base or none are). More information on PCTE processes and activities can be found in [6] and [1]. Table 2.2 on the following page lists some of the operations provided by PCTE for managing activities.

PCTE Activity Operations	
ACTIVITY_START	Starts a new activity in the current process
ACTIVITY_END	Ends the activity of the calling process, committing any outstanding modifications in the context of the enclosing activities
ACTIVITY_ABORT	Ends the current activity of the calling process, discarding or committing any outstanding modifications, in the context of the enclosing activities.
LOCK_RESET_INTERNAL_MODE	Resets the internal mode of a lock to its default value
LOCK_SET_INTERNAL_MODE	Promotes the internal mode of a lock.
LOCK_SET_OBJECT	Establishes, or promotes, a lock with a specified or default mode.

Table 2.2 Some of the PCTE Activity Operations

2.7 Implementations

The relevance or success of any standard depends to a large extent on the degree to which it is accepted and conformed to by industry. In recent years it has become evident that an open standard for integrating tools into SEEs is vital to the realisation of the full potential of CASE. PCTE has emerged as the foremost specification in this area.

The early use of PCTE has included the experimental demonstration and application of the PCTE approach to environment building in major national and international programs like the European Community's ESPRIT and, in the USA, the Department of Defence's DARPA (Defence Advanced Research Projects Agency) STARS (Software Technology for Adaptable Reliable Systems). The STARS programme was established to demonstrate three integrated SEEs on three real applications, to evaluate the benefits and using this evaluation, to increase software productivity, reliability and quality by integrating process management and re-use technology in leading-edge SEEs. STARS is based on industry standards (including POSIX, X Windows System/Motif and ADA) and an open environment architecture. That two of the three prime contractors (Boeing, IBM and Paramax) chose PCTE as their basis for their SEEs demonstrates the strength of PCTE as a technology standard.

NATO's Nations Special Working Group on ADA programming Support Environments used PCTE as the basis for their Portable Common Interface Set (PCIS). See Section 4.6.1. PCTE is a core component of the DoD I-CASE (Integrated Computer-Aided Software Engineering). Such was the interest in PCTE that in 1992 a US government/industry forum, the North American PCTE Initiative (NAPI), was established. NAPI was responsible for recommendations on extending the standard, producing a publicly available implementation of PCTE, establishing a PCTE validation capability for users by vendors and support for the acquisition of PCTE implementations and PCTE-based products in the USA. Early in 1994, the responsibilities of NAPI were taken over by the Object Management Group's PCTE SIG (Special Interest Group) [8]. It is already incorporated into the standards used by many organisations.

The interest in PCTE is not limited to the military. It has also been used as the foundation of commercially available SEEs from different vendors e.g. Émeraude (TRANSTAR (previously GIE Émeraude), France) Portos (EDS Scicon), PCTE/6000 (IBM), HPCTE (University of Siegen in Germany), Heuristix PCTE (Heuristix India), Verilog PCTE (Verilog France). Verilog PCTE, for instance, is a UNIX implementation built on the Oracle distributed RDBMS. Émeraude provides a UNIX and a WINDOWS implementation of PCTE. EAST (SFGL, France) and Enterprise II (Syseca, France) are integrated CASE environments which use the Émeraude implementation of PCTE as their foundation. In June 1994 Groupe Bull and Syseca (Thomson-CSF's ISV affiliate) announced a major consolidation of their application development framework business with the launching of a software (CASE) joint venture called TRANSTAR. TRANSTAR rely heavily upon the technology developed by GIE Émeraude in France[7].

The major multinational platform vendors have announced their commitment to PCTE in one way or another e.g. there has been work done at Digital to integrate PCTE into Digital's existing COHESIONworX framework [10] for CASE and to achieve a high level of interoperability with existing non-PCTE tools already integrated into the framework [11]. In April 1994 ICL agreed to distribute the Émeraude PCTE products; for example to vendors of commercial software tools, in order to assist porting to PCTE and to academic users for teaching the principles of open software engineering [8]. In January 1994 VISTA technologies announced support for ToolTalk in its product, PCTE Workbench; with ToolTalk extension, PCTE Workbench combines the potential of the two open standards: ToolTalk and PCTE [8].

2.8 Evaluation

PCTE has been very successful as a standard for a Public Tool Interface for integrated SEEs, this being evident from the diversity of platforms for which PCTE implementations are available, and the international support shown by its acceptance as an ISO standard in July 1994. This chapter has explored the concepts behind PCTE, demonstrating its strength as a data integration technology (with very limited control integration) suitable for portable CASE tools, while noting that, although it has a

strong object oriented flavour, it is not object oriented in the truest since it lacks a vital object oriented mechanism which would allow operations or methods to be associated with objects. It is with a view to enhancing its object orientation and the control integration of a PCTE environment that we wish to integrate it with OMG's Object Management Architecture (OMA).

In Chapter 3 we turn our attention to the OMA and the role of the Common Object Request Broker Architecture (CORBA) within this architecture, to explore the ideas behind OMA before further discussion on an integration between the two specifications.

CHAPTER 3 OMG CORBA

This chapter introduces the concepts behind the Object Management Group's specifications for the Object Management Architecture (OMA), and discusses why these specifications will have an important role to play in the evolution of distributed software technology. Section 3.2 outlines the goals which OMG hope to achieve. Section 3.3 gives an overall view of OMA and describes in detail the role of the Common Object Request Broker Architecture within OMA. The CORBA specification is the part of OMA in which we are most interested for the purpose of integration with PCTE. As a result it will be the main focus of the following sections. The structure of CORBA is outlined in Section 3.4, while the role of CORBA's Interface Definition Language is described in Section 3.5. The success of any standard depends on its industrial relevance and the support it receives from industry, that is the extent to which it is adopted. In view of this Section 3.6 analyses the support for OMA within the industry by outlining the available implementations of CORBA and discussing how it relates to other standards.

3.1 Distributed Computing

Strides in the advancement of technology, especially in telecommunications and workstation designs, and the advent of low priced personal computers are rapidly altering the traditional face of the computer industry. The advances involve new technologies, both in the way data is transmitted and in the way that communications are integrated with data processing capability. Distribution can be viewed as the computing paradigm of the future. This current drive towards distributed computing is prompted by the very real corporate demand that, if information is distributed throughout the organisation, then access to that information should also be distributed. The challenge for many

organisation today lies in the evolution from a centralised data processing architecture (reliance on mainframes) to a distributed architecture.

As well as providing an information processing environment better matched to the information needs of business, distributed systems are able to offer other advantages, mainly the potential for "openness" [39], i.e. reducing the restrictiveness of being tied to products of a particular manufacturer. However the price of this openness, and the dispersing of processing away from the mainframe and into the personal computer and workstation, is increased complexity. Also, since distributed systems typically evolve through the federation of heterogeneous independent systems, this determines a need for integration. Thus the primary evolution costs from centralised to distributed systems, as already stated, are not those of hardware, but are related to the quality, cost and lack of interoperability of software.

3.2 Object Management Group

The members of the Object Management Group (OMG), a consortium setting vendor-independent specifications for the software industry, have a shared goal of developing and using integrated software systems. The agreed criterion for a methodology for building such systems include the support of modular software production; it must encourage reuse of code; allow useful integration across lines of developers, operating systems and hardware; and enhance the long-range maintenance of that code. Members of OMG believe that the object-oriented approach to software construction best matches this criteria.

A more indirect end-user benefit of object-oriented applications, provided that they co-operate according to some standard, is that independently developed general purpose applications can be combined in a user-specific way.

OMG was founded in 1989 to "realise interoperability between independently developed applications across heterogeneous networks of computers, to help reduce complexity, lower costs, and hasten the introduction of new software applications. OMG plans to accomplish this through the introduction of an architectural framework with supporting detailed interface specifications. These specifications will drive the industry towards interoperable, reusable, portable software components based on standard object-oriented interfaces" [29].

OMG has defined an infrastructure for distributed computing called the Object Management Architecture (OMA). While industry has worked hard to provide a distributed model that allows users to be able to select their applications, networks, systems and services, no such model has yet matured [34]. The diversity of applications and platforms are making such systems increasingly difficult and complex.

The most popular approach to distributed computing has been that of the client-server. Client-server computing is a concept, about "breaking down large-scale system complexity into small, manageable parts; the problem is making the parts communicate with a single system view or interface" [34]. Distribution enabling technologies are often referred to as middleware, since they reside between the operating system and applications. Remote Procedure Calls, RPCs, are one such class of middleware. They function similar to normal programming calls, completing a single processing chore in a series of steps undertaken by a software program. RPCs separate the calling program and the called procedure into two processes. The calling program is the client; the called process is the server. To

accomplish this, you need to make the call in one process, communicate the input parameters to another process and get the procedure to execute in the remote process. RPCs have been around for quite a while, but the desire to build distributed, client/server applications in a networking environment has renewed the interest in them (RPCs). However the mass appeal of RPCs has been limited somewhat, due to the fact that they address only the communications aspect of distributed applications [58]. To address this limitation the DCE Remote Procedure Call service, which is the communications layer of the Open Software Foundation (OSF) Distributed Computing Environment (DCE), is designed to provide an integrated solution to distributed applications [58]. The OMA also addresses the whole area of distributed applications; however OMA/CORBA (the communications component of the Object Management Architecture) goes well beyond the RPC technique because it directly supports object oriented software [33].

3.3 Object Management Architecture

OMG has defined common terms, interfaces and a framework for distributed computing in the Object Management Architecture (OMA). In this framework, objects interact through an Object Request Broker (ORB). The OMA specifies the basic mechanisms that compliant applications must support to use an ORB, including how objects make requests and get responses, basic services provided to all objects, and facilities that are useful in many applications.

The Object Management Architecture has a broad notion of what constitutes an object. Objects are literally any element in the distributed system. An object can be an application, process, class or instance of a class. The only requirement is that the object supports an OMA-compliant interface. An object is referred to as an object

implementation in the OMA. The OMA specifies how these objects interact via an ORB[32].

In OMA, client objects make a request of an object implementation. A request is the invocation of an operation. The ORB then handles the request and any response to the client object. This can include dispatch and delivery of the request, synchronisation and delivery of any response or exception. Thus, the OMA is similar to the client/server model. The key difference is that the "Client" and "object implementation" (server) describe the roles that each object can exhibit. However a given object can take on either role for a particular interaction. The OMA is therefore more of a peer-to peer model [32].

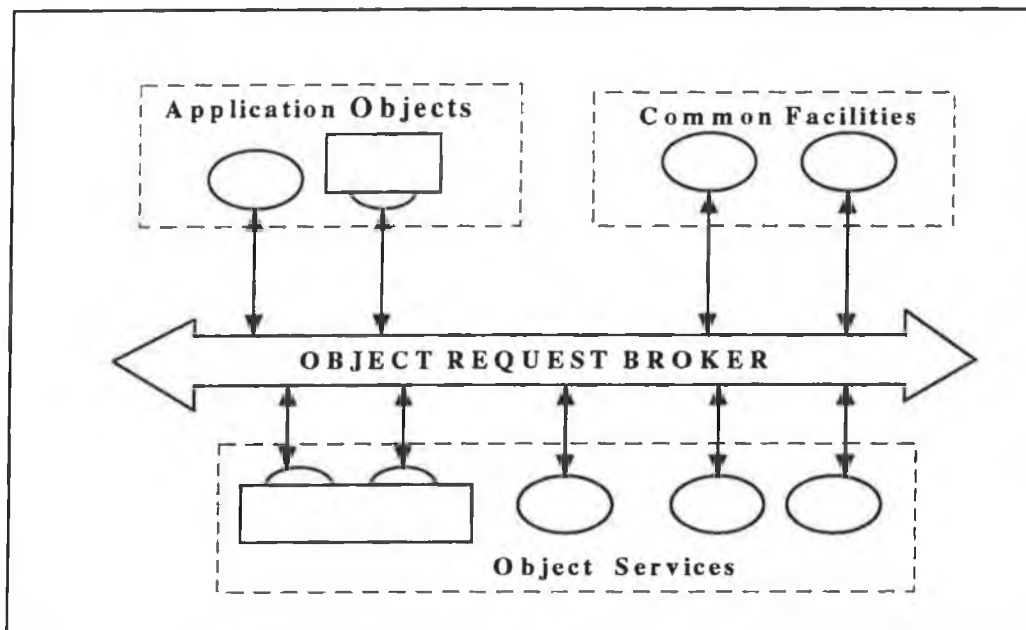

Figure 3.1 OMA Reference Model (See page 55 of [29])

Figure 3.1 shows the structure of the Object Management Architecture. The solid boxes represent software with application programming interfaces, while the dashed boxes represent categories of objects with object interfaces.

The **Object Request Broker** (ORB) enables objects to transparently make and receive requests and responses in a distributed environment, i.e. it is the communications heart of the OMA standard.

Object Services (OS) is a collection of services and object interfaces that provide basic functions for realising and maintaining objects (see Section 3.3.1).

Common Facilities (CF) is a collection of classes and objects that provide general purpose capabilities that are useful in many applications.

Application Objects (AO) are objects specific to particular end-user applications.

In general terms, the Application Objects and Common Facilities have an application orientation, while the Object Request Broker and Object Services are concerned more with the "system" or infrastructure aspects of distributed object management. Common facilities may, however, provide higher-level services- such as transactions and versioning- that make use of primitives provided within Object Services.

The three categories (OS, CF and AO) reflect a partitioning in terms of functionality, from those basic to most applications (or common enough to broad classes of applications to standardise) to those too application-specific or too standardised at this time. Common Facilities exemplifies a key concept that the OMA promotes: class reusability. Thus, the

Object Request Broker, Object Services and Common Facilities will be the focus of OMG standardisation efforts [29].

In general, Object Services, Common Facilities and Application Objects all intercommunicate using the Object Request Broker. An ORB provides "the basic mechanism for transparently making requests to - and receiving responses from - objects located locally or remotely without the client needing to be aware of the mechanisms used to communicate with, activate or store the objects" [5]. As such it forms the basis for building applications composed of distributed objects, and supporting interoperability between applications in homogeneous and heterogeneous environments. The interfaces to objects that communicate via the ORB are defined using the Interface Definition Language (IDL) included in the CORBA specification (See Section 3.5). Adherence to the Object Management Architecture will speed the design and delivery of robust applications that fit into an object-oriented environment. Applications can be viewed as collections of building blocks linked together at run time to complete various tasks [37].

3.3.1 Object Services

This section outlines the Object Services (OS) component of OMA. OS provide basic operations for the logical modelling and physical storage of objects [29], and as such it is of interest in this thesis, because it is to provide at least part of such functionality that we would wish to use the PCTE's OMS, in an integration between PCTE and the OMA. Object Services defines a set of intrinsic or root operations that all classes should implement or inherit. Objects do not have to use the implementation of basic operations provided by OS nor do objects have to provide all basic operations. For example, an object may provide its own data storage or an object that models a process may not

provide transactions. The operations provided by Object Services are made available through the ORB; however, they may also be made available through other interfaces. For example there may be additional interfaces that comply with non-OMG standards or that are optimised for higher performance. The operations that Object Services can provide include:

- *Class Management.* The ability to create, modify, delete, copy, distribute, describe and control the definitions of classes, the interfaces to classes, and the relationships between class definitions.
- *Instance Management.* The ability to create, modify, delete, copy, move, invoke and control objects and the relationships between objects.
- *Storage.* The provision of permanent or transient storage for large and small objects, including their state and methods.
- *Integrity.* The ability to ensure the consistency and integrity of object state both within single objects (e.g. through locks) and among objects (e.g. through transactions).
- *Security.* The ability to provide (define and enforce) access constraints at an appropriate level of granularity on objects and their components.
- *Query.* The ability to select objects or classes from implicitly or explicitly defined collections based on a specified predicate.
- *Versions.* The ability to store, correlate and manage variants of objects

The types of sub-components that could be used to implement Object Services include object oriented database management systems, or perhaps PCTE's OMS.

It is important to note that applications need only provide or use OMA-compliant interfaces (An OMA-compliant application consists of a set of inter-working classes and instances that interact via the ORB) to participate in the Object Management

Architecture. They need not themselves be constructed using the object-oriented paradigm. This is very useful when trying to integrate OMA with PCTE or when migrating from traditional systems to Object Orientation. Basically it means that part of your system may be implemented in a procedural language, but by "encapsulating" it in an IDL interface, it may be accessed by other OMA objects. This also applies to the provision of Object Services. For example, existing relational or object-oriented database management systems could be used to provide some or all of the Object Services.

The OMA assumes that underlying services provided by a platform's operating system and lower-level basic services, such as networking computing facilities, are available and usable by OMA implementations. Specifically, the Object Management Architecture does not address user interface support. The interfaces between applications and windowing systems or other display support are the subjects of standardisation efforts outside the OMG. Eventually, Common Facilities may provide standard user interface classes. In addition, the Reference Model does not deal explicitly with the choice of possible binding mechanisms [29].

3.4 CORBA

The Common Object Request Broker Architecture (CORBA) is the name given to the specification of the ORB component, it is designed "to allow integration of a wide variety of object systems" [30]. CORBA is a general solution to application integration, moving away from the conventional point-to-point solution. Generality of the architecture is provided by a high-level declarative language to describe objects, IDL (see Section 3.5).

The components of CORBA are clients, object implementations, the ORB and object adapters. A client is an entity that wishes to perform an operation on the object; the interface that the client can see is independent of where the object is located and what programming language it is implemented in. An object implementation is the code and data that actually implements the object, i.e. an object in itself. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. An object adapter is the primary means for an object implementation to access ORB services. Sections 3.4.1 - 3.4.4 describes these components briefly, see [30] for greater detail.

3.4.1 Structure of an Object Request Broker

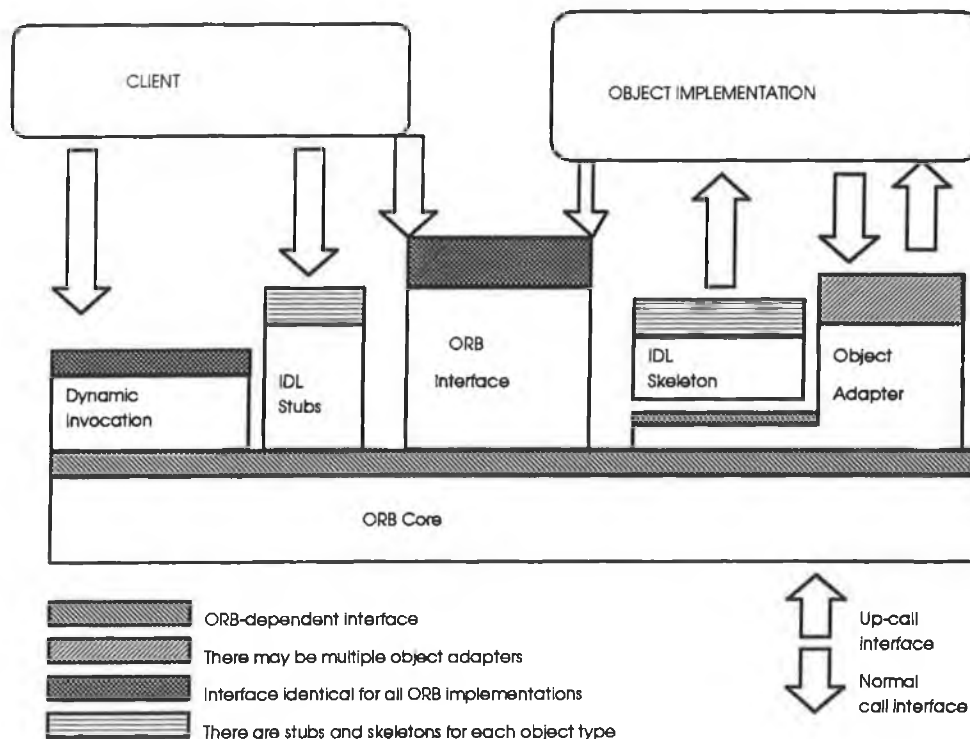

figure 3.2 The Structure of ORB Interfaces [30]

Figure 3.2 shows the structure of an individual Object Request Broker (ORB). The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. Operations that an object can provide are advertised to clients through the interface definition of an object. Definitions of the interfaces to objects can be defined in two ways, statically using the Interface Definition Language, IDL (see Section 3.5), or dynamically accessed by adding interfaces to the Interface Repository. To make a request, the client can use the Dynamic Invocation Interface or an IDL stub. When using the Dynamic Invocation Interface to make a request the same interface is used regardless of the interface of the target object. If an IDL stub is being used to make a request then a specific stub depending on the interface of the target object must be used. The receiver of the message is indifferent to which of these two methods is used.

The ORB locates the appropriate implementation code, transmits parameters and transfers control to the Object Implementation code via an IDL skeleton. The ORB may provide some services to the object implementation (through the object adapter during performance of the request) and directly to the client [30]. The object implementation receives a request as an up-call through the IDL generated skeleton. The ORB's functionality frees programmers from the details required by other application distribution methods.

3.4.2 Client

A client of an object has an object reference that refers to the object, and invokes operations on the object. A client is restricted to knowledge of the logical structure of the object provided by its interface and experiences the behaviour of the object through invocations. A client is a program or process initiating requests on an object. However it is important to remember that something is a client relative to a particular object, i.e. the implementation of one object may be the client of other objects. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

Clients access object-type-specific stubs as library routines in their program. The client thus sees routines callable in the normal way in its programming language. All implementations will provide a language specific data type to use to refer to objects, often an opaque pointer. The client then passes the object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation [30].

3.4.3 Object Implementations

An Object Implementation provides the actual state and behaviour of an object; figure 3.3 shows the structure of an object implementation.

An object implementation defines the following :

- methods for operations defined in the IDL interface; it also implements these methods.
- procedures for object activation and deactivation (usually).
- controls access to the object
- deals with object state persistence by using object or non-object facilities.

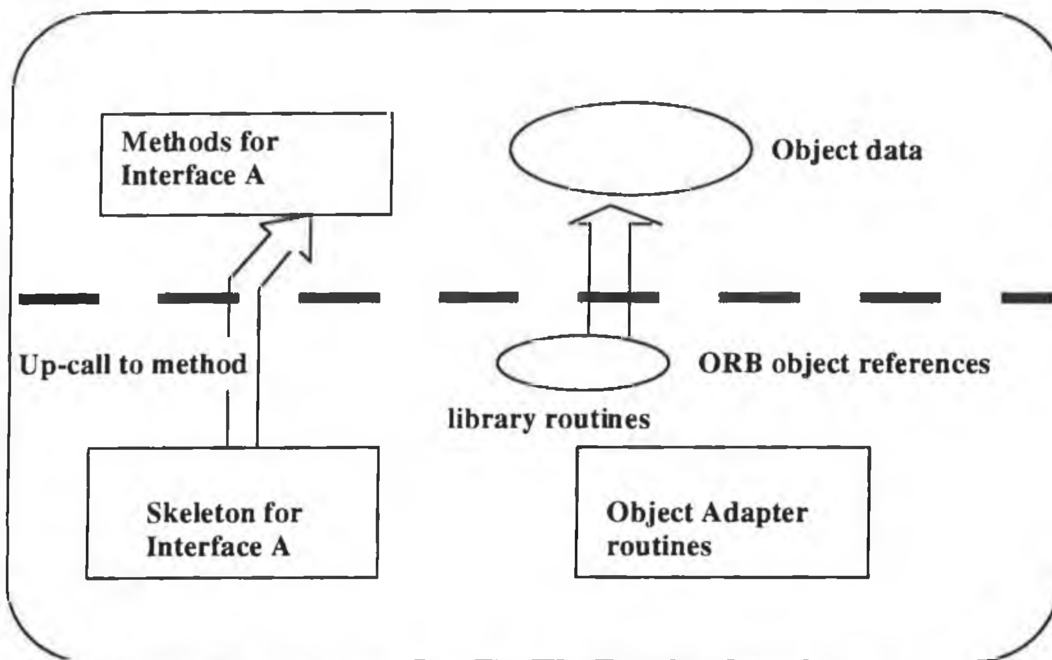

Figure 3.3 Structure of an Object Implementation [30]

3.4.4 Object Adapter

Object Adapters are "the primary way that object implementations access services provided by the ORB" [30] and are built on a private ORB-dependent interface. Object Adapters provide the following functionality :

- generation and interpretation of object references
- method invocation
- security of interactions
- object and implementation activation and deactivation
- mapping object references to the corresponding object implementations.
- registration of implementations

It is difficult for the ORB to provide a single interface suitable for all objects due to the large range of object properties (e.g. granularity, lifetimes, policies, implementation styles etc.). Through Object Adapters the ORB targets groups of object implementations that have similar requirements with interfaces tailored to them.

There are a variety of possible object adapters. However most object adapters are designed to cover a wide range of object implementations. For example the Basic Object Adapter (BOA), can be used for most ORB objects with conventional implementations., or the Object-Oriented Database Adapter(OODB) uses a connection to an object-oriented database to provided access to the objects stored in it. Since the OODB provides methods and persistent storage, objects may be registered implicitly and no state is required by the Object Adapter [30].

3.5 Interface Definition Language

IDL (the Interface Definition Language) is the language "used to describe the interfaces that client objects call and object implementations provide" [30]. The IDL, Interface Definition Language, defines the types of objects by specifying their interfaces. An

interface consists of named operations and the parameters to those operations. IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. Clients are not written in IDL, which is a purely descriptive language, but in languages for which mappings from IDL concepts have been defined. The mapping of an IDL concept to a client language will depend on the facilities available in the client language [30]. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

IDL obeys the same lexical rules as C++, its grammar being a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. IDL is a declarative language; it does not include any algorithmic structures or variables. The syntax for IDL is described in Appendix A of this thesis [30]. IDL Bindings already exist for C, C++ and ADA. An IDL specification consists of one or more type, constant, exception, interface (see Sections 3.5.1 - 3.5.5) or module definitions; examples in the Sections 3.5.1 - 3.5.5 illustrate particular aspects of IDL that will be referred to later in this thesis (Chapter 4). The module construct is used to scope IDL identifiers. It consists of the module keyword and one or more type, constant, exception, interface or other module declarations.

3.5.1 Interface Definition

An interface can contain one or more of the following elements: constant, type, exception, attribute or operation declarations (see following sections). An interface definition provides the basic framework for describing the objects manipulated by the ORB; it is the means by which a particular object implementation tells potential clients what operations are available and how they can be invoked. Therefore the constant, type, exception, and

attribute declarations contained with an interface specify the constants, types, exception structures and attributes exported by the interface. For example in the definition of the *env* interface, (see below), the interface specifies a constant integer called export which has the value of PROTECTED, a previously defined constant (defined using the pre-processor #DEFINE).

Operation declarations specify the operations that the interface exports (or offers). Operations declarations only take place within the context of an interface definition, and are explained in greater detail in Section 3.5.2. IDL interfaces have an optional inheritance mechanism whereby interfaces can be derived from other previously defined interfaces. In the following example the interface *env* inherits from a previously defined interface *pact_env*:

interface *env* : *pact_env* {
    const short int *export* = *PROTECTED* ;
    const short int *usage* = *NAVIGATE* ;
}

An interface which is derived from another interface may refer to elements of the base interface as if they were its own elements, as long as references to base interfaces are not ambiguous. A derived interface may also redefine any of the type, constant, operation, attribute and exception names which have been inherited from its base interfaces [30].

3.5.2 Operation Definition

Operation declarations in IDL are similar to C function declarations. They describe the services which an object implementation can provide through its interface to potential clients. The following section explains the semantics behind the syntax of an IDL operation definition, see Appendix A for the syntax of IDL.

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked.
- The type of the operation's return result; the type may be any type which can be defined in IDL; operations which don't return a type must specify the void type.
- An identifier which names the operation in the scope of the interface in which it is defined.
- A parameter list which specifies zero or more parameter declarations for an operation. A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed, these being **in**, **out**, **inout** which mean the parameter is passed from client to server, from the server to the client, and in both directions respectively.
- An optional raises expression which indicates which exceptions may be realised as a result of an invocation of this operation [30].

Take for example the interface *compiler* shown below: it includes the operation declaration for *compile*, which advertises a *compile* operation available from the *compiler* object to its potential clients; the object implementation for the *compiler* object will provide an implementation for this operation. The operation declared within this interface

is to have two parameters *objectname* and *params*; both are strings to be passed from the client to the server (notice the "in string"), *objectname,* specifying the object to be compiled and *params* specifying the compiler parameters.

```
interface compiler {
        readonly attribute COMPILER_RESULT errors ;
        void compile(in string objectname, in string params);        };
```

3.5.3 Attribute Definition

An attribute declaration within an interface is logically equivalent to declaring a pair of accessor functions, one to retrieve the value of the attribute and one to set the value of the attribute. The optional readonly keyword indicates there is only one accessor function, i.e. the retrieve value function. Take for example the attribute *errors* which is declared in the *compiler* interface in section 3.5.2; errors is declared here as a readonly attribute of the enumerated type *COMPILER_RESULT,* this implying that the implementation of this interface will require a function of the same name as the attribute(i.e. *errors),* which will return the value of this attribute *errors*, see Appendix D for the implementation code of the compiler interface.

3.5.4 Enum & Type Declaration

IDL provides constructs for naming data types, i.e. C-like **typedef** declarations that associate an identifier with a type. The basic types supported by IDL are float, double, long, short, unsigned long, unsigned short, char, boolean, octet and any.

Enumerated types consist of ordered lists of identifiers, the identifier following the **enum** keyword defining a new legal type, *COMPILER_RESULT* in the example shown below. Enumerated types may also be named using a **typedef** declaration.

```
enum COMPILER_RESULT {    COMPILER_FAILED,
                          COMPILED_OK,
                          COMPILED_WITH_ERRORS,
                          COMPILER_NOT_TRIED };
```

Refer to the example of the *compiler* interface in Section 3.5.2 to see how such an enumerated type may be used.

3.5.5 Constant Definition

IDL provides a Constant Definition for associating a constant with an identifier. The constant can be any of the following types: integer, char, floating point, boolean, string or scoped name [30]. An example of this is shown in the interface *env* example in Section 3.5.1.

3.5.6 Exception Declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request, basically a method of error handling. Each exception is characterised by its IDL identifier, an exception type identifier and the type of the associated return value [30]. If an exception is returned as the outcome to a request then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised. If no members are specified, no additional information is accessible when an exception is raised [30]. In addition to a standard set of exceptions that may be signalled by the ORB, operation specific exceptions can be specified using the raises expression.

3.6 Implementations & Industrial Relevance

The OMG does not unilaterally develop standards; its members agree to adopt specifications and provide compliant products. The reliance on commercial technology and the fact it is completely "open" (any company can join the OMG or submit technology in the specification process) ensures the relevance of the standard and the capability to quickly move the industry to a common architecture for distributed computing [32].

Most of the major vendors now support CORBA or are CORBA-compliant, demonstrating CORBA's growing strength in the market place- e.g. SunSoft provide a CORBA implementation called Distributed Objects Everywhere (DOE) which is available fully integrated for UNIX and Solaris 2.0. Hewlett Packard have developed their own native CORBA implementation called ORB PLUS; their Distributed Smalltalk is also

CORBA compliant; both are available on UNIX platforms. AT&T's Co-operative Frameworks is a CORBA compliant ORB with some object services. DEC have developed a CORBA implementation called ObjectBroker (formerly called ACAS); ObjectBroker is available on MacOS, Windows, NT, UNIX and VMS platforms. An Irish company called IONA Technology provided the first implementation of the CORBA standard called ORBIX which is available for most major platforms, including Sun, HP and Windows NT. This variety of implementations endorses the CORBA specification's place within the industry. The relationship between the OMA and other standards is an important consideration for end users. The Object Management Group works with other standard groups through common members and a liaison committee [31], including the PCTE SIG (see section 1.2).

Another important consideration, when evaluating the place of a technology within the industry, is to look at the strength of vying technologies. Until recently Microsoft had been pushing OLE as the major competitor to CORBA. OLE (Object Linking and Embedding) is an application integration framework that supports compound documents and sharing of objects between applications. Distributed OLE provides these capabilities across a network. However both OLE and Distributed OLE are still only available for PCs (running Microsoft Windows) and Window's Workgroups[36]. OLE and Distributed OLE are based on the Component Object Model (COM) which has both significant similarities and differences to the OMG Object Model. The most significant difference is that COM provides for application integration through the definition of a binary format for an object interface. Applications can interoperate as long as the objects adhere to this format. In contrast, the OMA uses IDL as an intermediate language to describe the interfaces that objects support: applications are integrated through the use of standard interfaces and the ORB, and no restrictions are placed on implementation as is the case with COM. Other differences include the fact that COM does not support inheritance between object interfaces, and COM supports the notion of a guaranteed unique object

identifier (Guaranteed Unique Identification (GUID)). This assumption differs from the OMG model where objects can have multiple object identifiers (OIDs) [32].

An ObjectWorld conference opinion poll in 1994 [31] found CORBA to be the more favoured model. These factors, and an industry demand for CORBA-compliant applications, have precipitated a move towards joint interoperability of COM and CORBA. This will be addressed in the next version of the CORBA specification (CORBA 2.0). OLE Interoperability will take the form of an OLE/CORBA object adapter which will allow for Object references that are intrinsic to COM to be resolved and allow for activation and execution of a CORBA object [35].

3.7 Conclusion

In this chapter we have seen the role of CORBA in OMA, which was designed to ease the development of integrated software systems, and in particular the importance of IDL within the CORBA structure in order to achieve this integration. Independently developed applications which adhere to the OMA specification can be combined in user specific ways. This is the beauty of OMA, it reduces the complexity of distributed systems.

The fact that OMG does not unilaterally develop standards(rather its members agree to adopt specifications and provide compliant products), as well as its reliance on commercial technology and the fact that any company can join the group or submit technology to the specification process ensures, in my opinion, the relevance of the OMA specifications. The number of CORBA implementations and compliant products readily available on the market demonstrates the endorsement of the CORBA specification by the computer industry.

Now that we have looked in detail at both the PCTE and Object Management Architecture/CORBA specifications, we will examine the relationship between them in the following chapter, as well as how they may be integrated and what the benefits of their short term integration would be.

CHAPTER 4 INTEGRATING PCTE AND CORBA

This chapter outlines the different approaches to the short term integration of PCTE and OMG CORBA taken during the research, and discusses why such an integration was deemed attractive. As previously stated in Chapter 1 work on the integration of PCTE and OMG CORBA into a single standard is already in progress. This convergence of PCTE and CORBA may take a considerable amount of time. Meanwhile both specifications could be used together to their mutual benefit. Substantial benefits can be gained by integrating the current specifications so that they can be used together immediately. The purpose of the integration strategies discussed in this chapter is to provide a viable short term approach to the integration of PCTE and CORBA, without altering either of the existing standards, prior to their convergence.

We begin by looking at the issues which make such an integration desirable. The relationship between PCTE and OMG's CORBA is a potentially complementary one[1]. Section 4.1 examines the relationship between PCTE and OMA (of which CORBA is a component), emphasising the areas in which they are potentially complementary, and examines how these may be best harnessed to the advantage of each, without altering either standard.

Given the complementary nature of the relationship between the two specifications, the objectives for their integration are discussed in Section 4.2. The remainder of this chapter contains an outline of the two different approaches taken during this research to finding a short term solution to the convergence of PCTE and CORBA. Sections 4.3 and 4.4 introduce the language mapping of DDL to IDL (and vice versa) as an integration strategy, which was the favoured initial approach to integration, but was deemed

unfeasible by research. This language mapping was the favoured approach because its success would have ensured a direct translation from CORBA objects to PCTE objects. Even though this thesis proves this approach unfeasible, we include it for the valuable lesson of why such a mapping is unfeasible and its implications for the future convergence of PCTE and OMA/CORBA. The alternate integration strategy, the definition of IDL interfaces for PCTE tools, which demonstrates the considerable benefits of an interim integration between CORBA and PCTE, is introduced in Section 4.5 and discussed in greater detail in Chapter 6.

4.1 Relationship of PCTE and OMA

This section reviews the information provided in Chapters 2 and 3, which is of particular relevance to the compatibility of PCTE and OMA, in particular CORBA. It contains a description of the features of each specification which may be used to complement each other. The relationship between PCTE and OMA is discussed here as opposed to the relationship between PCTE and CORBA, partly to give the more global picture, and partly because the benefits of integrating PCTE and CORBA come from the fact that CORBA is a component of OMA (See Section 3.3). Therefore what is said here of OMA applies equally to CORBA.

In [1] the OMG PCTE SIG describe a number of possible relationships which may exist between OMA and PCTE. The two specifications could be used together in a coexisting, layered, or complementary manner. Since the interest of this thesis rests in their short term integration, the emphasis in the remainder of this section will be placed on the complementary nature of their relationship. Sections 4.1.1 and 4.1.2 outline the primary

features of both specifications, Section 4.1.3 goes on to discuss how these features may be used for their complementary integration.

4.1.1 Primary Features and Strengths of PCTE

The following points summarise the primary features and particular strengths of the Portable Common Tool Environment (PCTE) specification.

- Data integration for CASE environments in which tools (i.e. programs) create and access shared data objects (repository).
- An Object Management System (OMS) providing transparent access to data objects in a distributed standardised repository running over heterogeneous platforms.
- Facilities for transparent process distribution (process modelled as objects).
- APIs for object and repository management functions.
- Supporting APIs for tool portability across operating systems [1].

Thus PCTE has a strong sense of data integration provided by its data modelling mechanisms (data objects, links and attributes) and its Data Definition Language (DDL) for schemas, see Section 2.4. PCTE provides multiple views of the object base using dynamic working sets and decentralised distributed Schema Definition Sets (SDSs), see Section 2.3. Its security model prevents unauthorised data access. PCTE provides a nested transaction model, a mechanism for ensuring consistent data, as well as enforcing integrity constraints. PCTE also provides a concurrency control model (locks) and versioning facilities.

4.1.2 Primary Features and Strengths of OMA

The following points summarise the primary features and particular strengths of the Object Management Architecture (OMA) specifications, of which the CORBA specification is a component:

- Horizontal enabling technology with an extensible architecture supporting applications that are collections of interoperating, co-operating distributed objects (data and methods) [1].
- An Object Request Broker (ORB) "provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems"[2].
- A set of Object Services - basic services for creating and maintaining objects, this being the framework on which application interoperability is based.
- A set of Common Facilities - this is a set of general purpose objects and classes that may be useful in many applications

Thus the OMA specification supports full object orientation, operations on objects, i.e. methods as well as a data interface to object "contents". It provides support for fine grain objects, i.e. high-speed access plus low storage overhead. The overall performance of OMA is high because of the following features: persistent object references, no built in integrity constraints, fine grain execution management and local object optimisation. OMA provides multiple object adapters which are specialised "drivers" for different flavours of object implementations and object systems.

OMA also provides object services for distributed application portability and interoperability. Common functionality in different applications (such as storage and retrieval of objects, mailing of objects, printing of objects) [2] is realised by OMA's Common Facilities which provide general purpose capabilities which are useful in many applications.

4.1.3 Complementary Standards

Having reviewed their individual strengths, let us now discuss the integration of PCTE and OMA, in a global sense, and how this integration can be of benefit to both specifications. As we have seen in the previous sections there is overlap between PCTE and OMA; the potential exists for their combined differing approaches to be complementary in the following areas:

While PCTE, through its strong notion of data integration, does support the notion of objects, it is not object oriented in the true sense, in that PCTE objects are data objects, i.e. they have state but do not have behaviour. This lack of true object orientation and the lack of support for low storage overhead/high speed access to fine-grain objects could be addressed by integrating PCTE technology with OMA-based products or in the long term evolving PCTE to become OMA conformant (i.e. the role of the OMG PCTE SIG).

As stated earlier, PCTE components are integrated using mainly data integration. OMA could be also be used to provide tighter integration of the environment, by providing improved control integration.

PCTE was especially designed to meet the needs of CASE environments by providing rich data modelling facilities. To address the specific needs of CASE environments, OMA specifications could be extended or added so as to incorporate PCTE functionality such as data modelling, enforced integrity constraints and support for configuration management. Part of the OMA's Object Services specification is that of the provision of a persistent store for OMA objects. The PCTE object base could be used to provide such a persistent store as required by, but not yet available for, OMA object implementations [1]. At the moment the only OMA component which is fully specified and implemented is the CORBA component; yet PCTE and the concepts behind it could be very useful in the development of the Object Services component of the OMA. The nature of the data and the (inter-data) relationships in a CASE environment is very complex. The PCTE OMS incorporates a complex object model, semantic data model theories (see [18],[23]), as well as making use of database system technology, in order to allow these complex relationships to be modelled in an intuitive way. Therefore using the semantically rich data modelling provided by PCTE's OMS to implement at least part of the Object Services component of OMA could be very valuable.

PCTE and OMA have a complementary relationship and so their convergence is an attractive proposition already undertaken by the OMG PCTE SIG. This thesis is concerned with the provision of an integration strategy for PCTE and OMA (in particular CORBA) which can be used in the interim until their eventual convergence, and it proves that such short term integration has many benefits to offer particularly to PCTE.

4.2 INTEGRATION STRATEGIES

The integration strategies discussed in this chapter indicate ways in which PCTE and CORBA might be used together in a mutually beneficial way without any specification changes, based on the information contained in section 4.1. What we hope to achieve by this research is a viable short term integration of PCTE and CORBA which would from a PCTE developer's point of view :

- Use CORBA (IDL and object requests) to enhance control integration between PCTE tool components. PCTE normally relies on data integration; therefore CORBA could be used for stronger integration between PCTE tool components, also facilitating the composition of PCTE tools.
- Tool components would be encased in IDL interfaces. The tools themselves would continue to store and share data in the PCTE repository but, by having an IDL interface, would be able to interact with the ORB and avail of all OMA services and other OMA compliant systems.
- Use CORBA to associate behaviour with PCTE purely data objects in order to make PCTE objects object oriented in the fullest sense.

From a CORBA object implementor's point of view, the integration would hope that :

- PCTE could be used as a persistent service to store the state of objects. The PCTE API would be used directly in the object implementation's code and would not be visible outside of the object itself [1].

As stated earlier, one of the prerequisites of these integration strategies was that they were to require no changes to the existing specifications. We acknowledge the effectiveness of the data integration facilities provided by SDSs (using DDL) for Computer-Aided

Software Engineering tools in a PCTE repository, and wish to combine this with control integration which can be provided by CORBA, in order to arrive at a more fully integrated Software Engineering Environment.

In brief, we wish an integration strategy to arrive at an IDL interface definition for a PCTE tool, allowing the tool to become in effect an object which can avail of the ORB and other OMA Object Services. A strategy should allow the PCTE tool to behave as if it were a CORBA application which is able to interoperate with other CORBA applications on different machines and seamlessly interconnect with multiple object systems. In turn, we wish to avail of the rich data modelling underlying the PCTE OMS in order to allow the PCTE repository to be used as a persistent store for CORBA objects.

Two approaches to developing an interim integration strategy were explored during the course of this research. The first approach taken was that of a direct language mapping of PCTE's Data Definition Language, DDL, to OMG CORBA's Interface Definition Language, IDL (and vice versa). This language mapping approach to integration was the most attractive proposition because if it had been successful a simple translation tool would have automatically generated IDL definitions from PCTE definitions, and allowed a direct translation from CORBA objects to PCTE objects. The mapping of DDL to IDL was envisaged as an approach to allow PCTE tools to be "encased" in an IDL interface, so that they could be viewed as OMA/CORBA objects with access to the ORB and other OMA facilities, making them truly object oriented, while also increasing the control integration between them and facilitating the development of composite PCTE tools, see figure 4.1. Section 4.3 describes briefly the mapping of DDL to IDL, while Chapter 5 describes in greater detail how DDL language constructs may be mapped to IDL language constructs. Chapter 5 also contains a discussion of how this thesis proved such a direct language mapping, given the current specifications of DDL and IDL, was not feasible for

the short term integration of PCTE and CORBA, and what future extensions are required to DDL in order to make such a mapping a feasible integration strategy.

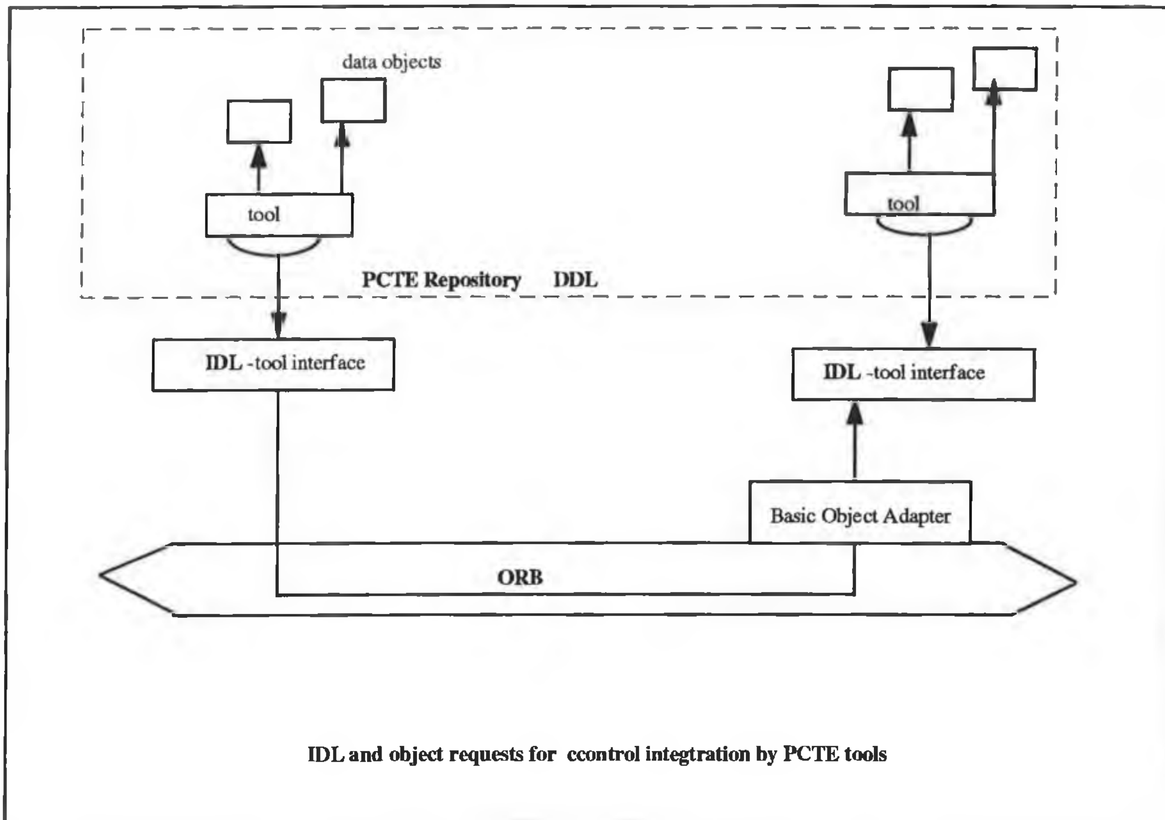

Figure 4.1 *Mapping DDL to IDL*

The reverse language mapping, that of mapping the IDL to DDL was initially seen as an approach to allow OMA objects to be defined and exist in the PCTE repository, in this way using the PCTE repository as a persistent store for OMA objects, see figure 4.2. However preliminary research found the mapping of IDL to DDL to be unfeasible for a number of reasons including incompatible scoping rules and the fact that DDL contains no notion of the concept of operations attached to objects(i.e. PCTE objects are not compatible with the definition of the OMG Object Model outlined in [2], see Section 4.4).

Having demonstrated that a direct language mapping was not possible given the current specifications of DDL and IDL, another route to integration was sought. The second approach to integration explored was the definition of IDL interfaces for PCTE tools. PCTE tools are stored as static contexts objects within the repository, a static context being an object which contains the program code of a PCTE tool. A language mapping of DDL to IDL, if successful, would have provided an automatic integration of PCTE and CORBA, allowing a direct translation from DDL to IDL.

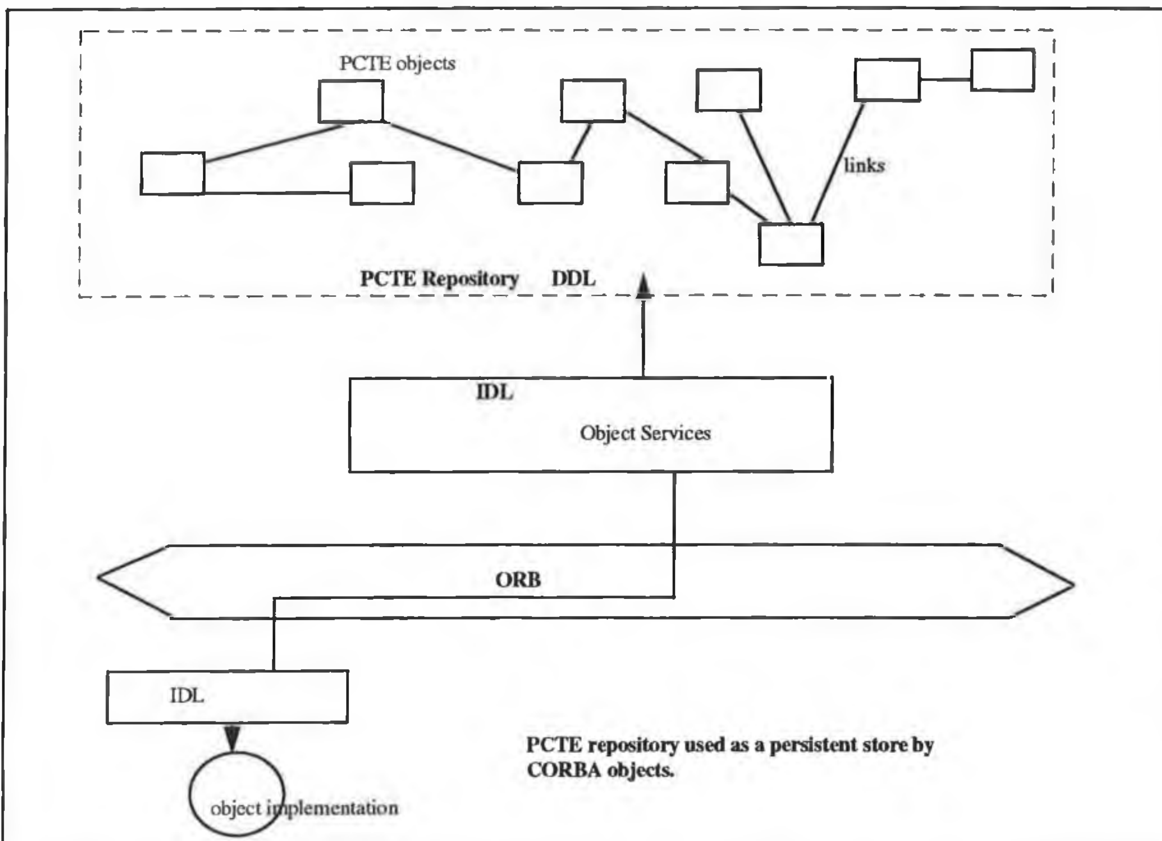

Figure 2 Mapping IDL to DDL

The definition of IDL interfaces for PCTE tools does not provide such an "automatic" integration, being more of a methodology for integration. This approach provides a

beneficial integration for PCTE. By employing this approach PCTE objects become fully object oriented; they can be viewed as CORBA objects by wrapping them in an IDL interface, thus allowing them access to the ORB and other OMA facilities. The definition of IDL interfaces for PCTE tools facilitates greater control integration between PCTE tools and the development of composite PCTE tools. This second approach is outlined in Section 4.5 and is described in greater detail in Chapter 6. The syntax of the DDL and IDL language constructs described in these sections are given in Appendix A ([5]) and Appendix B ([3]) respectively.

Now that we have discussed what we wish an interim integration strategy to achieve, we now turn our attention to how these aims are to be achieved.

4.3 Mapping DDL to IDL

This section describes the concepts involved in mapping PCTE's DDL to CORBA's IDL. The motivation for such a mapping has already been discussed in Section 4.1. As stated earlier in Chapter 2, DDL is a formal notation for defining types, and is used to define the types in the four standard PCTE SDSs (Schema Definition Sets). Typing is a prominent characteristic of the PCTE data model, such that every instance in the PCTE repository belongs to a defined type. It places restrictions on the properties of PCTE entities which are created and managed in terms of their specific type. Typing is the fundamental element which allows the data integration of tools. The PCTE data definition language, DDL, is a formal notation for defining these types[4].

This mapping aimed to provide an equivalent IDL definition for a DDL definition of a tool's working schema (i.e. the tool's view of the repository). DDL definitions are

composed of sequences of SDSs. In the mapping DDL to IDL developed, each tool was to have its own IDL interface. Each SDS in the tool's DDL definition would be mapped onto a separate interface which may then be inherited by the tool's IDL interface and therefore accessed by it. Thus the mapping places a great deal of importance on the inheritance mechanism for interfaces in IDL. DDL objects, links and attributes would be mapped to IDL interfaces. There is a number of reasons for this, one of which is to facilitate the importation of types from one SDS to another, allowed by DDL. Since DDL objects, links and attributes can all be imported into other SDSs and used within the SDS possibly to be extended with other properties, i.e. type-in-SDS, therefore, by defining IDL interfaces for DDL objects, links and attributes, these IDL interfaces can be inherited by the IDL interface mapping of any other SDS which imports them.

Another reason for the definition of IDL interfaces for attribute declarations is to facilitate the fact that DDL attributes can be declared and then applied to an object or a link. Interfaces which were mapped from DDL attribute type declarations will be inherited by the interfaces representing objects or links to which the attributes apply. Each SDS mapped onto an IDL interface will inherit from the following interfaces: any interface which is a mapping of a type importation required by the SDS or a mapping of an object, link or attribute declaration (or extension, in the case of object and link) contained in the SDS.

Chapter 5 describes the language mapping in greater detail, describing how the DDL language constructs- for example, type importation declarations, object declaration, link declarations- are mapped into IDL. It also explains why the mapping of DDL to IDL is fundamentally flawed by the fact that, since IDL interface definitions define operations which an object's implementation will provide to its clients, and DDL models only data with no concept of behaviour or operation, the IDL interface mapped from DDL are

meaningless. However, by extending DDL, a meaningful mapping would be possible, Chapter 5 also contains a description of the necessary extensions to DDL.

4.4 MAPPING IDL TO DDL

As stated earlier, the language mapping of IDL to DDL was initially seen as an approach to allow OMA objects to be defined and exist in the PCTE repository, thus using the PCTE object base as a persistent store for OMA objects. This proved unfeasible for two very important reasons. PCTE objects are not compatible with the OMG Object Model, outlined in [2], mainly because there is no mechanism for associating PCTE objects with tools. Another reason for the unfeasibility of this approach is that IDL scoping rules are incompatible with DDL scoping rules. IDL syntax allows for nested declarations of interfaces. This characteristic is accomplished through the following rules taken from the syntax given in [3]:

```
<definition>  ::=   <type_dcl>     ";"
       |            <const_dcl>    ";"
       |            <except_dcl>   ";"
       |            <interface>    ";"
       |            <module>       ";"

<module>      ::=   "module"    <identifier>    "{"
                                <definition>+   "}"
```

The rules shown above allow IDL modules and its interfaces to be nested within other modules. In contrast, PCTE's SDSs are linear in nature[3], and therefore unable to model the possibly nested IDL interfaces and modules.

This conflict in scoping rules makes a reverse mapping, the mapping of IDL to DDL, impossible. However in a mapping of DDL to IDL, this does not affect the mapping other than the fact that the IDL nested scoping feature would not be utilised; linear PCTE SDSs would be mapped onto IDL definitions, which are not nested.

Having briefly discussed the reasons why a language mapping approach to the integration of PCTE and OMA cannot be successful until alterations have been made to the both specifications (much of the work being done by the OMG PCTE SIG involves these very alterations [12]), we see that an alternative approach to integration is necessary, since it was a pre-requisite at the outset of this research to provide an interim integration which will not necessitate changes to either specification, and so now we turn to the second approach explored in this research, the definition of IDL interfaces for PCTE tools.

4.5 IDL Interfaces for PCTE Tools

This section outlines briefly the definition of IDL interfaces for PCTE tools as an integration strategy. This strategy was explored after research had found that a mapping of DDL to IDL would necessitate the altering of the PCTE standard, in particular DDL, in order to make it feasible as an integration strategy.

To allow CORBA access to PCTE tools, they must have an IDL interface. This IDL interface advertises the services its object implementation provides to potential clients (i.e. CORBA objects which wish to avail of the services advertised). Therefore the object implementation of an IDL interface advertising operations of PCTE tools must contain in their object implementation some method of executing the PCTE tools within the object base, see Figure 4.3. The method used in this research was to embed a PCTE shell script (as opposed to a UNIX shell script) within the CORBA object implementation. The shell script acts as a wrapper or buffer between the CORBA object implementation and the PCTE tool. It also allows us to use the PCTE activity operations (See Section 2.6) as provided by the PCTE API (Application Program Interface) to ensure that the object base remains in a consistent state. Chapter 6 describes the definition of IDL interfaces for PCTE tools as an integration strategy in greater detail.

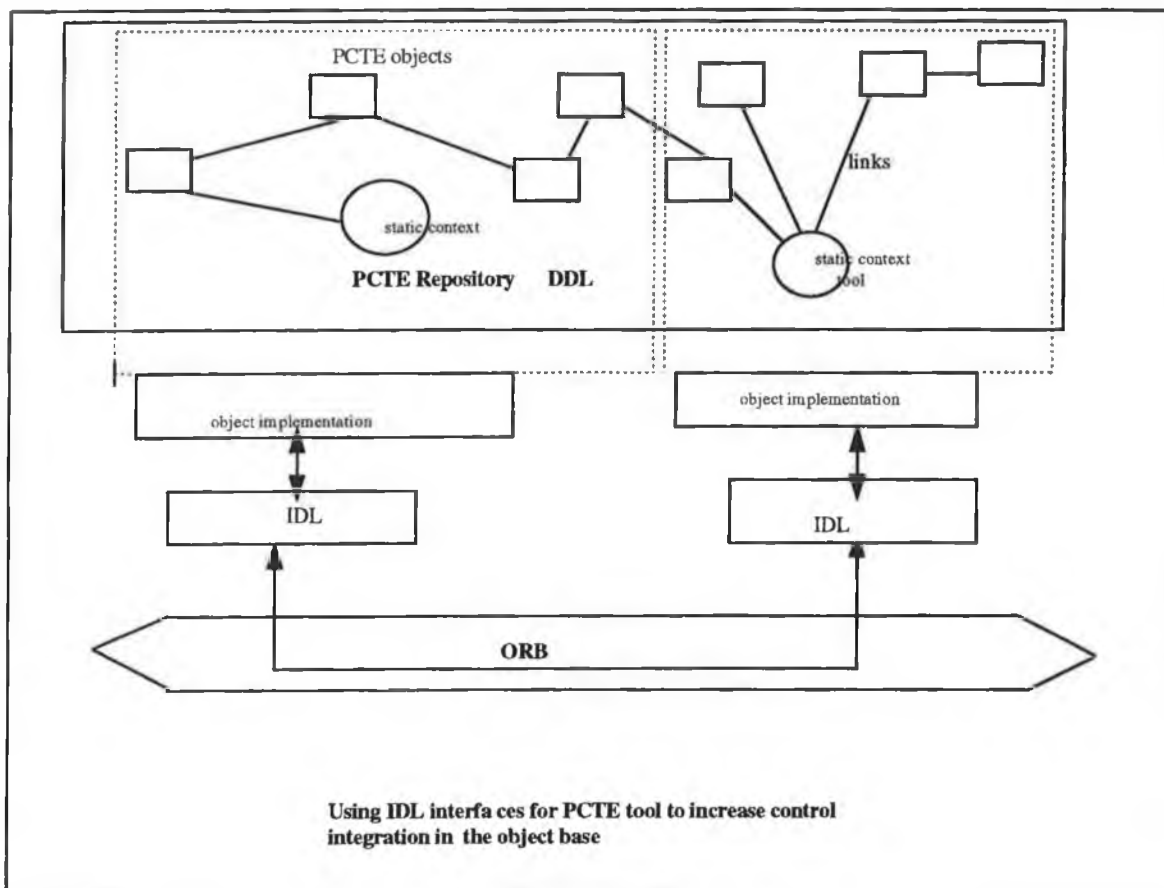

Figure 4.3 Defining IDL interfaces for PCTE tools

This approach to the integration of PCTE and OMA is not a mutually beneficial integration since it offers nothing to CORBA. The benefits of the integration are to PCTE alone, since it does not allow the persistent storage of OMA objects in the PCTE repository. However the benefits which this approach provides to PCTE makes such a compromise acceptable, these benefits including increased control integration between the tools in the PCTE repository, support for the full object orientation of PCTE objects (by defining an IDL interface for a PCTE, the tool's behaviour as well as its data can be modelled), support for the composition of PCTE tools, and access to other OMA service and other OMA compliant systems. Now that the approaches to integration taken during research have been introduced, before we examine them in greater detail we will take a look at some related works, and see where they fit in with this thesis.

4.6 Related Works

In order to evaluate where the research presented in this thesis fits in relation to other research in the field, it is necessary to look at how it differs from other related works: PCIS (Portable Common Interface Set), COHESIONworX/PCTE (Digital) and OOTIS (IBM AIX-CASE). These are described in the following sections.

4.6.1 PCIS

PCIS was founded by the NATO Special Working Group on ADA Programming Support Environments in 1991. The project's goal was to identify a SEE framework based on PCTE (see [1], [2]), Syseca's Enterprise-II environment [21] and other available

standards. The PCIS project is introduced and specified in [45] and [46] respectively. As stated previously PCIS is based on the PCTE model, its architecture being the same as a PCTE one. Based on analysis and evaluation of IRAC's Object Management System requirements [48] and the NIST/ECMA Reference Model object management services [21], PCIS is designed to enhance PCTE. Among the framework services areas that would supplement PCTE, as identified by [16] and [47], are object oriented services, fine-grained management of data, user-managed data, trigger services, life cycle process services and tool integration and co-ordination [12].

In contrast to PCTE which supports medium and large grain data, PCIS also supports fine grained data. PCIS provides a new definition language, PCIS Interface Definition Language, (PIDL) in order to support fine granularity and further object orientation, openness, integration and co-operation among tools [12]. These requirements of fine granularity, openness, improved integration etc. demanded a framework that supports not only the data sharing provided by PCTE but also behaviour sharing among tools [16]. PCIS fulfils these requirements.

PCTE's inheritance and object identity facilities encouraged the PCIS project members to use object oriented database answers to accomplish the behaviour sharing facilities which are missing in PCTE. PIDL language is based on both PCTE's Data Definition Language(See Section 2.4) and the interface definition language defined by CORBA (see Section 3.5) [12]. It is this fact that PCIS is basically a hybrid of PCTE and CORBA as well as other technologies that differentiates it from the research contained in this thesis, where both specifications are used unaltered.

4.6.2 COHESIONworX/PCTE

This section describes work done at Digital in order to provide an implementation of CORBA based on the PCTE standard, the objective of which was to enhance the usability of PCTE through the addition of a high level object oriented interface [49]. This work has provided a partial implementation of CORBA (only the dynamic interfaces), which is based on the ECMA PCTE standard. The project relied purely on ECMA PCTE facilities for OMS access, process start-up and interprocess communication; so this implementation respects ECMA PCTE semantics of the dynamic context for child processes, guaranteeing that CORBA servers conform to PCTE security and activity semantics. This was made possible by using Digital's implementation of CORBA, ACAS [50] and exploiting the two tier architecture of ACAS: there is an upper layer that implements the CORBA semantics and a lower layer that interacts with the OS and the network. The lower layer was re-implemented using PCTE facilities and supporting the upper layer for process execution, data access security, distribution and interprocess communication. The CORBA specification does not place any restrictions on how it is to be implemented and so, because the COHESIONworX/PCTE approach to integrating PCTE and CORBA exploits a particular aspect of the ACAS implementation of CORBA, we cannot assume that this approach would work for all CORBA implementations.

The CORBA implementation delegates the process activation and deactivation to PCTE and all CORBA server objects are activated by means of PCTE primitives, thus respecting the semantics of the dynamic context for child processes (e.g. activities and security) see Section 2.5. The use of PCTE for process distribution supports application start-up anywhere in a PCTE distributed environment. Communication between CORBA objects relies exclusively on PCTE messages queues, as this ensures a transparent and secure

exchange of data between CORBA applications running in a distributed PCTE environment.

Digital have also worked on a project to integrate PCTE into Digital's existing COHESIONworX Framework for CASE and to achieve a high level of interoperability with existing non-PCTE tools already integrated into the framework [11]. COHESIONworX is an open framework that offers a distributed software development environment based on: distributed control services, a graphical desktop environment, and a set of integrated development tools [11]. The control integration aspect of this project is based on the CORBA implementation using PCTE services. The introduction of CORBA as one of the integration technologies of an SEE, such as COHESIONworX, achieves two important results from the perspective of the tool integrator and framework builder. It ensures semantic integrity among PCTE SDSs, and it allows the definition of tool interfaces, so that they can make their services available to the rest of the environment, while hiding the implementation details.

Work is currently in progress to engineer a full-compliant CORBA/PCTE implementation based on the new Digital's product ObjectBroker 2.5 that implements OMG's ORB. This version will also allow integration with Microsoft OLE 2 via the COM protocol [52]. The success of these projects have proved that the ECMA PCTE is indeed the right base on which to build O-O extensions. Further research at Digital[51] confirms that PCTE and CORBA set the stage for the addition of other services needed by the CASE framework provider and tool integrator-for example ATIS (A Tool Integration Standard) version and configuration management services- in order to achieve a robust and flexible framework for CASE tool integration.

4.6.3 OOTIS Tool Integration Model

Object Oriented Tool Integration Services (OOTIS) Tool Integration Model is an architecture for a CASE Tool Integration platform addressing both data and control integration and covering the performance spectrum from coarse to fine grained integration. It integrates an Object Oriented control sharing model with an extended PCTE data sharing model [53]. OOTIS extends PCTE in two ways. It provides, first of all, support for object oriented control integration and, secondly, support for fine-grained objects.

OOTIS permits the definition of operations applicable to object types. These definitions specify interfaces (signatures) only [54]. OOTIS also permits the definition of tools that provide implementations of operations, and mappings that specify which implementations are to be used in which circumstances. Programs can invoke operations on specific objects in either of the two convenient ways specified by the OMG CORBA. A dispatcher generated by OOTIS from the tool definitions will route each invocation to the appropriate implementation(s).

The details of the tool definitions are such that they can be combined using composition operators, allowing separately-written tools to be composed easily. The OOTIS control integration support thus introduces CORBA-compliant object oriented method resolution and tool composition into PCTE. OOTIS control integration is modelled by three SDSs. One extends the pre-defined PCTE SDS, **metasds**, with interface definitions, consisting of operation type definitions and associations of interfaces with object types. The other two SDSs model tools, consisting of implementations and mappings. Therefore the OOTIS approach to control integration also involves changes to the PCTE specification.

4.7 Evaluation

All of the related works described in the previous section had been undertaken before the formation of the OMG PCTE SIG and the commitment to the convergence of PCTE and OMA. The PCIS and OOTIS projects both require alterations to the PCTE specification, while the success of the COHESIONworX/PCTE project highlights the benefits of utilising the complementary nature of the relationship between CORBA and PCTE. However, as pointed out earlier, it relies heavily on a particular implementation of CORBA. We are interested in an integration strategy for CORBA which is independent of implementation.

The OMG PCTE SIG is committed to the convergence of PCTE into the OMA/CORBA specifications. The purpose of this thesis was to find an integration strategy which can be used in the interim to support this convergence, since both specifications have much to offer each other. For instance, part of OMA, the Object Services specification, which is not yet complete, is the specification of a persistent store for OMA objects; the PCTE repository could be used as such a store. The PCTE OMS is suitable for such a purpose because the focus of the PCTE specification is on data integration and as such it provides an elegant and powerful data modelling system. Even though specifically designed for CASE environments, the complexity of the relationships in such environments means that the PCTE OMS has evolved to a position where it can model complex data and relationships for other environments. Likewise, CORBA could be used, as described earlier, to enhance PCTE to full object orientation, increasing the control integration between PCTE tools, and to facilitate tool composition. For these reasons a language mapping between PCTE's DDL and CORBA's IDL appeared very attractive, as it would have allowed the direct mapping of PCTE objects to OMA/CORBA objects. However,

the research in this thesis shows that such a language mapping is not possible without altering the specification of DDL. The extensions which DDL would require in order to be compatible with IDL are described in Section 5.4.

Having demonstrated that such a mapping is unfeasible, the definition of IDL interfaces for PCTE tools was explored as an integration strategy which aimed to improve control integration between PCTE tools and enhance PCTE objects to full object orientation. However this strategy does not cater for the mapping of CORBA objects to the PCTE repository. Thus the definition of IDL interfaces for PCTE tools is a much weaker integration strategy than would have been provided by the mapping of DDL to IDL (and vice versa), had such a language mapping been successful.

However, apart for compromising the benefits of a mutual integration, it does achieve the other objectives for an interim integration strategy. This approach increases the control integration between PCTE tools. If the IDL interface is used to access the tool then, to all clients, the tool seem truly object oriented because the object implementation of the interface encompasses both the tool's data and the static context (executing tool), its behaviour. Also this approach facilitates the development of composite PCTE tools. Therefore it has much to offer PCTE as an integration strategy while waiting for the convergence of the two specifications.

In Chapter 6 we will examine the strategy for defining IDL interfaces for PCTE tools in greater detail. First, in Chapter 5, we discuss the language mapping of DDL to IDL and why it was proven by this thesis to be unfeasible as an integration strategy.

CHAPTER 5 LIMITATIONS OF THE MAPPING OF DDL TO IDL

This thesis proves that the mapping of PCTE's Data Definition Language (DDL) to CORBA's Interface Definition Language is unsuccessful as an integration strategy for the current specifications of DDL and IDL as they stand, and therefore it is unsuitable as an interim integration strategy which requires the unaltered specifications for both PCTE and CORBA to be used. In order to understand the limitations of the mapping, let us first look at the motivation and general concepts behind the development of such a mapping, and in turn the form that this mapping would take. Section 5.1 outlines the basic ideas involved in the mapping of DDL to IDL, while the details of how DDL language constructs are mapped to IDL language constructs are described in Section 5.2.

Initially this mapping of DDL to IDL appeared to be the most attractive integration strategy, because it would allow the automatic generation of an IDL interface from a DDL definition via a simple language translation. By generating such IDL interfaces from the DDL definitions, it was hoped that these IDL interfaces would "encase" PCTE tools, and integrate them with the OMA structure, allowing them to avail of the ORB in order to increase the control integration between the tools in the PCTE repository. However the mapping proved to be unfeasible for this purpose. Section 5.3 discusses why this thesis proved this approach not viable as an interim integration strategy. DDL must be extended for compatibility with IDL, in order to facilitate meaningful mappings between these two languages, a description of the necessary extension to DDL being contained in Section 5.4.

Before looking at the precise details of how DDL language constructs can be mapped to IDL, we will begin by looking at the general concepts behind the mapping.

5.1 GENERAL MAPPING CONCEPTS

As stated earlier in Chapter 2, DDL is a formal notation for defining types, and is used to define the types in the four standard PCTE SDSs (Schema Definition Sets). Typing is a prominent characteristic of the PCTE data model, such that every instance in the PCTE repository belongs to a defined type. It places restrictions on the properties of PCTE entities, which are created and managed in terms of their specific type. Typing is the fundamental element which allows the data integration of tools. The PCTE data definition language, DDL, is a formal notation for defining these types[4].

This mapping aimed to provide an IDL definition for DDL definition of a tool. DDL definitions are composed of sequences of SDSs. In mapping DDL to IDL, each tool has its own IDL interface, and each SDS in the tool's DDL definition is mapped onto a separate interface which may then be inherited and therefore accessed by the tool's IDL interface. Take for example a PCTE tool, a C compiler we will call *ccomp,* whose working schema consists of the **sys**, **env**, **c_prog** and **pact** SDSs. Then the inheritance specification for the IDL interface of this tool will contain at least **sys**, **env**, **c_prog** and **pact** as its base interface as shown below.

interface *ccomp* : sys, env, c_prog, pact {
...... };

This mapping places great importance on the inheritance of interfaces. DDL objects, links and attributes are mapped to IDL interfaces within the mapping, since it must be

possible to import all three into other SDSs, i.e. allow their IDL interfaces to be inherited by other interfaces. Interfaces which are mapped from DDL attribute type declarations are inherited by the interfaces representing objects or links to which the attributes apply. Take for example the IDL interface *name* as given in Section 5.2.2, and the following link type declaration taken from the c_prog SDSs, see Appendix C.

h : **composition link** (<u>*name*</u>, *subname*)
to include_file;

The inheritance specification for the IDL interface of the link type, *h*, would be as follows:

interface *h* : *name* {
...... } ;

Each SDS mapped onto an IDL interface, inherits from the following interfaces: any interface which is a mapping of a type importation required by the SDS or a mapping of an object, link or attribute declaration (or extension in the case of object and link) contained in the SDS.

Because we treat links as objects (since we allow the definition of IDL interfaces for links) for the mapping, we need to address the existence of links with cardinality greater than one. This means that references need to be materialised either to the link in its complete plurality, enabling access to all of the instances of the link, or a single instance of the link. The concept of the link as a plurality, or set of link references, is inherent in the notion of links that anchor relationships that are not one to one relationships. Although link is a simple name it potentially refers to many instances of the same type, a set of link instances[6]. The DDL to IDL mapping handles the notion of sets of link instances by

allowing multiple associations to be stored by link and object type interfaces as arrays of pointers to objects which satisfy the associated interface types. In the case of object type interfaces, this means an array of pointers to object types which will satisfy each of the interface types, an array for each link type. In the case of link type interfaces, it means arrays of pointers to objects which will satisfy the appropriate object interface types, an array for each PCTE object type.

Each DDL data type (boolean, natural, integer, float etc.) is mapped onto an IDL data type. Constants such as WRITE, READ, PROTECTED, NAVIGATE etc. will be defined as symbolic constants in IDL. Sections 5.2.1 - 5.2.6 describe the language mapping in greater detail, describing how the DDL language constructs such as type importation declarations etc. are mapped into IDL. The examples attached to each section are taken in part from [13] but they have been altered to show how additional features of each construct are mapped. The syntax of the DDL and IDL language constructs described in these sections are given in Appendix A ([5]) and Appendix B ([3]) respectively.

5.2 MAPPING DDL CONSTRUCTS TO IDL CONSTRUCTS

This section describes how the DDL language constructs- i.e. type importation, attribute type, link type, object type, link type extension and object type extension declarations- are mapped onto IDL language constructs.

5.2.1 Mapping Type Importation Declarations

A type importation declaration in DDL (see Section 2.4.1) is mapped onto an IDL interface declaration where the interface's inheritance specification contains the name of the type to be imported, and the interfaces identifier is the type's local name within the SDS to which it is being imported. The type mode declaration (which shows how the type may be used, e.g. navigated or read, and if it can be exported from the current SDS) is represented as a constant declaration of identifiers called export and usage set to the appropriate value (PROTECTED, READ, WRITE, DELETE, CREATE, NAVIGATE) within the interface definition. This interface is then included in the inheritance specification of the IDL interface of SDS for which the type is being imported. Take for example the DDL type importation declaration shown below (explained in Section 2.4.1).

import object type *pact-env* **as env (usage navigate ; export protected) ;**

This can be mapped to the following IDL interface :

interface env : pact_env {
    const short int *export*= *PROTECTED* ;
    const short int *usage* = *NAVIGATE* ;}

5.2.2 Mapping Attribute Type Declarations

A DDL attribute type declaration (see Section 2.4.6) is mapped to an IDL interface, since it must be possible to import attributes into other SDS, i.e. in terms of the mapping, to allow SDS interfaces to inherit them. The type mode declaration would be mapped in the same fashion as described in Section 5.2.1. The interface contains a constant declaration

of type boolean identified by 'non_duplicated' which is set to TRUE or FALSE depending on whether the DDL keyword non_duplicated is present in the attribute type declaration or not.

The DDL value type indication clause is represented as attribute type declaration of an identifier named 'value' within the interface. The DDL initial value clause (indicating the initial value of the attribute), if present, is mapped onto a constant declaration, identified by *initial_value*. Take for example the following DDL attribute type declaration:

name **: (usage create ; export protected) non_duplicated string :=** "*John Smith*"**;**

This can be mapped to the following IDL interface.

interface *name* {
  const short int *export = PROTECTED* ;
  const short int *usage = CREATE* ;
  const boolean *non_duplicated = TRUE*;
  const string *initial_value* = "*John Smith*";
  attribute string *value*;
}

Note : An attribute definition in IDL is logically equivalent to declaring a pair of accessor functions, one to retrieve the value of an attribute and one to set the value of the attribute.

5.2.3 Mapping Object Type Declarations

A DDL object type declaration (see Section 2.4.2) is mapped to an IDL interface declaration. The IDL interface inheritance specification of an object declaration which contains a **child type of** clause (i.e. a declaration of a PCTE object which is derived from the objects specified after the child type clause) will specify the parent interfaces from which the object is to inherit. The IDL interface definitions of any attributes that the DDL object type contains will also be included in the inheritance specification.

The type mode declaration (see Section 5.2.1) of the object type is represented as a constant declaration of export and usage modes set to the appropriate values (PROTECTED or CREATE) within the IDL interface definition. The contents clause of the object type declaration is mapped as a type definition for a void pointer called "contents" within the interface declaration; this pointer can later, in the implementation of the interface, be set to an object of an appropriate type depending on the contents type. e.g. **file, pipe, device, audit_file** or an **accounting_log**.

The DDL component clause of the object type declaration (groups together objects which are related to each other) is represented as an array of pointers to objects, objects which satisfy the IDL link interface types mapped from DDL link types specified in the clause. An array exists for each link type in the clause, the MAX_SIZE of the arrays defined to be the maximum number of links allowable. If the component indication list of the component clause contains a link type declaration, the link type declaration is mapped to another interface (handled similar to the attribute type declaration within the link type extension in Section 5.2.6), and an array of object pointers is set up to hold references to all links of this type. Take for example the following DDL object type declaration:

c_source : **child type of** *source_file* **with**

contents **file ;**

attribute *name* ;

link *tool* ;

end *c_source* ;

This can be mapped to the following IDL interface :

interface *c_source* : *source_file*, *name* {

void * *contents* ;

tool links_1[*MAX_SIZE*] ;

}

5.2.4 Mapping Object Type Extension Declarations

A DDL object type extension (see Section 2.4.5) is mapped to an IDL interface definition where the IDL interface inheritance specification of the object type extension will specify the interface of the object being extended, as an interface from which to inherit. The interface declarations of any attributes that the object extension contains will also be included in the inheritance specification. The link and component clauses will be mapped in the same way as those within an object type declaration mapping (See Section 5.2.3). Take for example the following DDL object type extension declaration:

extend object type *project* **with**

attribute *name* ;

link *product* ;

component *current_projects* ;

end *project* ;

This can be mapped to the following IDL interface :

interface *X_project* : *project*, *name* {

product *link_1*[*MAX_SIZE*];

current_projects *component_1*[*MAX_SIZE*];

}

5.2.5 Mapping Link Type Declarations

A DDL Link type declaration (see Section 2.4.3) is mapped to an IDL interface declaration, where the interface's inheritance specification will contain the interface mapping of any attributes which apply to the link.

The type mode declaration (see Section 5.2.1) of a link is represented as a constant declaration of export and usage modes set to the appropriate value (PROTECTED, NAVIGATE, DELETE, CREATE) within the IDL interface definition. The interface contains a constant declaration of type boolean identified by 'exclusive' which is set to TRUE or False depending on whether the DDL keyword exclusive is present in the definition or not.

The interface contains a constant declaration of type boolean identified by 'non_duplicated' which is set to TRUE or FALSE, depending on whether the DDL keyword non_duplicated is present in the DDL definition or not. The interface contains a constant declaration of type short int identified by *stability* which is set to an appropriate value (ATOMIC, COMPOSITE or NONE) depending on whether the link is defined as being of atomic or composite stability or unstable. A link is stable if its designation object cannot be modified or deleted as long as this link exists. The interface contains a constant declaration of type short int identified by *category_name* which is set to an appropriate value (COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT or DESIGNATION) depending on whether the link is defined as being of link type *composition* (defining the destination object of the link as a component of the origin object), *existence* (keeps the destination object in existence as long as the link exists), *referential* (guarantees the existence of an object that can be referred to by a path name), *implicit* (used to reverse links of the other link categories when the reverse part of a relationship does not need to express any particular properties) or *designation* (relevant only to the origin object, they represent dynamic relationships) link.

If the DDL link type declaration contains a cardinality clause then two constant declarations of type short int, identified by *upper_bound* and *lower_bound* and set to take on the upper_bound and lower_bound cardinality of the link, are made within the IDL interface. The DDL key list clause is mapped to the IDL interface as an array of object pointers to objects which satisfy the IDL interfaces, mapped from the attributes which make up the key. A reverse link clause within a link declaration is mapped onto a link type name pointer declared within the IDL interface.

The "to" clause of the link declaration is mapped as an array of pointers to objects which satisfy the IDL interfaces of objects to which the link may point (an array exists for each

object type in the clause). The MAX_SIZE of the arrays is defined to be the maximum number of links allowable. Take for example the following DDL link type declaration :

subprog **: (usage navigate ; export protected) exclusive**

  non_duplicated

  composition link (*name*, *subname*) **to** *program*

with

  attribute *name* ;

end *subprog* **;**

This can be mapped to the following IDL interface :

interface *subprog* **:** *name* **{**

  const short int *usage* = *NAVIGATE* **;**

  const short int *export* = *PROTECTED* **;**

  const boolean *exclusive* = **TRUE ;**

  const boolean *non_duplicated* = **TRUE ;**

  const short int *category_name* = *COMPOSITION* **;**

  program to_1[*MAX_SIZE*]**;**

}

5.2.6 Mapping Link Type Extension Declarations

A DDL link type extension (see Section 2.4.4) is mapped to an IDL interface definition where the IDL interface inheritance specification will specify the IDL interface of the link being extended, as an interface from which to inherit. The interface declarations of any

attributes that the DDL link type extension contains will also be included in the inheritance specification.

The "to" clause is mapped in the same way as the "to" clause in a link type declaration is mapped (See Section 5.2.5). Take for example the following DDL link type extension declaration :

extend link type *tool* **to** *sctx*
with
attribute
    user **: string ;**
end *tool* **;**

This can be mapped to the following IDL interfaces: an interface for the link type extension and an interface for the attribute type declaration sub-component. The attribute type declaration is mapped to the following IDL interface :

interface *user* **{**
    const boolean *non_duplicated* **= FALSE ;**
    attribute string *value* ;
}

The DDL link type extension is then mapped to the following IDL interface :

interface *X_tool* **:** *tool*, *user* **{**
    sctx to_1[*MAX_SIZE*]**;**
}

5.3 LIMITATIONS OF THE MAPPING

Having examined the form that a mapping of DDL to IDL takes, let us now discuss why the usefulness of the mapping described in the previous sections is limited and unfeasible as an interim integration strategy.

As stated earlier, the purpose of generating an IDL interface from DDL definitions is to *encase* or *wrap* PCTE tools in an IDL interface, which would integrate them into the OMA structure allowing them to avail of the ORB in order to increase the control integration between the tools in the PCTE repository, to become full object oriented having both behaviour and state, and to allow PCTE tools to be activated using CORBA. In CORBA an IDL definition of an object defines the operations which the object can provide, the purpose of IDL being to provide a definition of objects based on the services or functions these objects can provide to their clients. In IDL operation declarations (see Section 3.5.2) are used to advertise to clients the services which the object can provide, and so the presence of operation declarations in an IDL interface definition is necessary to define the behaviour of the object.

From the description of the mapping in Section 5.2, we see that none of the DDL language constructs can be mapped to an IDL operation declaration. Also, looking closely at the examples of IDL interfaces generated from DDL SDSs in Sections 5.2.1 - 5.2.6, we notice a distinct absence of operation declarations in these interfaces. Even though the interfaces in these examples are all legal IDL syntax, their usefulness or meaning is limited, somewhat like a function definition which contains only variable

declarations. This is illustrated further below, where we demonstrate that, when each of the DDL language constructs (one of each type is taken as an example) used in the c_prog SDS are mapped to IDL. the resulting interface for the c_prog SDS contains no operation declaration. For a complete listing of the c_prog SDS, see Appendix C.

```
sds      c_prog :
import sys-name as name;
release:      integer      :=    1 ;
c_source      :      subtype of file ;
.....
tests    :      composition link
                to testests ;
.....
end c_prog ;
```

Using the mapping described in Section 5.2 we would get the following IDL definitions:

```
interface release {
        int     initial_value :=    1 ;    };
interface c_source : file {};

interface file : sys-file {};

interface tests {
        const short int category_name = COMPOSITION ;
        testsets to_1[MAX_SIZE];
} ;
```

interface *c_prog* **:** *file***,** *c_source***,** *tests***,** *release* **{}**

We saw in Chapter 2 that the entire PCTE repository, including the static context object which contain the PCTE tools (in either source code or executable form) is defined using DDL SDSs. Therefore from the above example we can see that the mapping of an SDS which defines a PCTE tool will also result in an IDL interface which contains no operation declarations, and therefore, while such an interface may be a valid IDL interface, it cannot advertise the functions or operations provided by this tool.

Therefore the IDL interfaces generated by the mapping of DDL to IDL described in earlier sections of this chapter cannot be used to increase control integration among PCTE tools or to activate PCTE tools using CORBA because these IDL interface do not advertise any operations or service for potential clients of these objects to avail of. Remember control integration is "the capacity to request operations from other tools in the system" [43].

Having ascertained that, while it is possible to map DDL language constructs to IDL, the resulting IDL interfaces are meaningless because none of these interfaces contain any IDL operation declarations. In order to understand why this incompatibility exists between DDL and IDL, we must remember that IDL models the behaviour of objects. This is its primary purpose, it does not model the data on which the object "behave". In direct contrast to this, PCTE's DDL is a data definition language used to define data types, which does not include any concept of function/operation or the behaviour of these strictly data objects. The lack of a mechanism for defining behaviour for DDL objects is the reason that DDL and IDL are incompatible, and therefore the reason why a mapping of DDL to IDL as an interim integration strategy is fundamentally flawed.

To overcome this incompatibility DDL must be extended to incorporate behaviour for PCTE objects. Because this would require changes to the PCTE specification it is unsuitable for the purpose of this thesis; however a future a mapping between the two languages is still attractive for integration purposes. It would however be incorrect to assume that the extensions to DDL, to be described in Section 5.4, would make DDL and IDL equivalent. The IDL interfaces mapped from an extended DDL would not be able to capture all the semantic richness of the data modelling provided by DDL. This is because DDL's data modelling relies heavily on the notion of types (object, link and attribute types), while CORBA has no notion of type, and so some of the semantic richness would be lost in the translation. The following section discusses the additional constructs that DDL would require in order to facilitate a complete mapping of DDL to IDL, and outlines some of the work already being carried out in this area.

5.4 EXTENDING DDL FOR COMPATIBILITY WITH IDL

This section describes the extensions that DDL would require before a useful mapping to IDL would be possible. In order to accomplish such a mapping DDL must be augmented with additional features which allow it to describe behaviour for data objects. This could be achieved by adding a mechanism for defining interfaces and operation signatures within DDL, as well as a mechanism for attaching or associating these interfaces with PCTE objects. As this would require changes to the PCTE specification, it is beyond the scope of this project. However let us look briefly at some of the progress being made in this area.

As already mentioned the PCIS project (described in Section 4.6.1) is based on the PCTE model. Therefore its architecture is the same as the PCTE one. However, the members of

the PCIS group analysed and evaluated IRAC's Object Management System requirements (see [8]), as well as the Object Management System services in the NIST/ECMA Reference Model (see [9]). This resulted in framework services supplements to the existing PCTE ones. One of these was the provision of a new definition language, PIDL. PIDL, PCIS Interface Definition Language, was needed for more integration and co-operation among tools than was provided by PCTE. These needs, as [10] states, demanded a framework that supports not only data sharing but also behaviour sharing among tools. PIDL is a language mutually based on PCTE's DDL and CORBA's IDL, incorporating features from each: as such, it is of significant interest when deciding what additional features DDL require in order to make it compatible with IDL.

The PIDL designers had to augment four DDL rules to provide for the connection of interfaces with object types in the Schema Definition Set, (SDS). For instance, the clause category, which provides the fundamental components within a SDS, was enhanced to allow a parameter type declarations[7]. In addition, the DDL's object type declaration and object type extension categories were augmented with an interface indication list to "associate a set of operation signatures with the object type in SDS"[10]. Lastly, the importation of parameter types from one SDS to another was accomplished through the expansion of DDL's category import type with parameter types and constants. An example can be found in [10].

The current work described in (Section 1.2) by the OMG PCTE SIG [12] involves the definition of these extensions to DDL and the wider effect of this on the PCTE standard. The OMG PCTE SIG regard the following areas as those necessary for consideration when extending DDL for compatibility with IDL, as included in their work to extend PCTE for complete object orientation.

- Interface Representation, where the description of how the interface hierarchy is represented in the metabase, i.e. which part is described in SDS and which at the intrinsic level. It is important to observe that this part of the model is described as an extension of the metasds, while the remaining two parts below are extensions of the system SDS [49].
- Method Implementation Representation, where the description of how tools and the methods they implement (static contexts, loadable modules or scripts) are represented at the metalevel and how they can be represented at the application level.
- Method mapping to interfaces, which describes the general way in which interface operations are mapped to implementations

Thus we see that extending DDL for compatibility with IDL is a viable proposition, and that an extended DDL will be included in that future specifications of OO PCTE [49]. However the purpose of this thesis is to find an integration strategy suitable for the current specifications of CORBA and PCTE, and so we must abandon the idea of using a mapping of DDL to IDL for the moment.

5.5 EVALUATION

In this chapter we have examined the concepts behind a mapping of PCTE Data Definition Language (DDL) to CORBA's Interface Definition Language (IDL). We have seen why such an integration strategy was initially seen to be so attractive, because it would allow the automatic generation of an IDL interface from a DDL definition via a simple language translation. By generating such IDL interface from the DDL definitions, the aim was that these IDL interface would "wrap" PCTE tools, and integrate them with the OMA

structure, allowing them to avail of the ORB in order to increase the control integration between the tools in the PCTE repository.

This chapter also explained that the mapping proved to be unfeasible for this purpose, given the currently specified DDL, even though it was possible to map from DDL language constructs into IDL language constructs. The problem arose because none of the DDL language constructs mapped to an operation declaration, arising from a basic incompatibility between the PCTE object model and the OMA object model. Objects modelled by DDL (i.e. PCTE objects) have no behaviour; therefore when these objects are mapped onto IDL interfaces, the corresponding interface has no operations defined, this defeating the purpose of defining and IDL interface. Much of the current work of the OMG PCTE SIG is concerned with extending DDL so that it is compatible with IDL. However until such extensions are made to DDL, the mapping of DDL to IDL cannot be used as an integration strategy.

This research was committed to finding an integration approach which could be used with the current specifications. Therefore having proven that the mapping of DDL to IDL is not possible with the current specifications (a valuable lesson in itself), the language mapping strategy to integration of PCTE and CORBA had to be abandoned in favour of an alternate route, defining IDL interfaces for PCTE tools, which will be the focus of the following chapter.

CHAPTER 6 IDL INTERFACES FOR PCTE TOOLS

As discussed in Chapters 4 and 5 the initial approach to integration for this research, that of a language mapping for DDL to IDL, proved unfeasible as an interim integration strategy because, in order to be successful, it would require alterations to the specification of PCTE, to DDL in particular. Therefore in this chapter we turn our attention to another approach to providing a strategy for the short term integration of PCTE and OMA'S CORBA. This chapter describes such a strategy, the definition of IDL interfaces for PCTE tools, illustrated by examples using the Emeraude PCTE V12 implementation and IONA Technology's ORBIX version 1.1 as the CORBA implementation. Some aspects of this chapter are specific to these implementations, for instance the Emeraude shell script, but there does exist an equivalent facility in other implementations (e.g. .bat files in Windows or DOS environments); so this method is portable with minor adjustments to other implementations and environments.

Section 6.1 outlines the general concepts behind the definition of IDL interfaces for PCTE tools as an integration strategy. Section 6.2 demonstrates with an example how to define an IDL interface for a PCTE tool. Section 6.3 describes how to implement such an IDL using a PCTE tool and shell script, while Section 6.4 discusses how this strategy facilitates the development of composite PCTE tools, and increases the amount of control integration in a PCTE environment.

6.1 General Concepts

An IDL interface must be defined for a PCTE tool to allow CORBA to access it, since it is through defining an IDL interface for objects that they can advertise the services they provide (see Section 3.5), thus making their services available to the whole environment, see figure 6.1. IDL interfaces are completely independent of implementation, this being the purpose of object implementations (Wherever object implementation is mentioned in the remainder of this chapter it can be taken to mean a CORBA object implementation as described in Section 3.4.3). In other words the IDL interface defines what services are available and how they may be invoked, while the object implementation defines how these are provided(i.e. the implementation details).

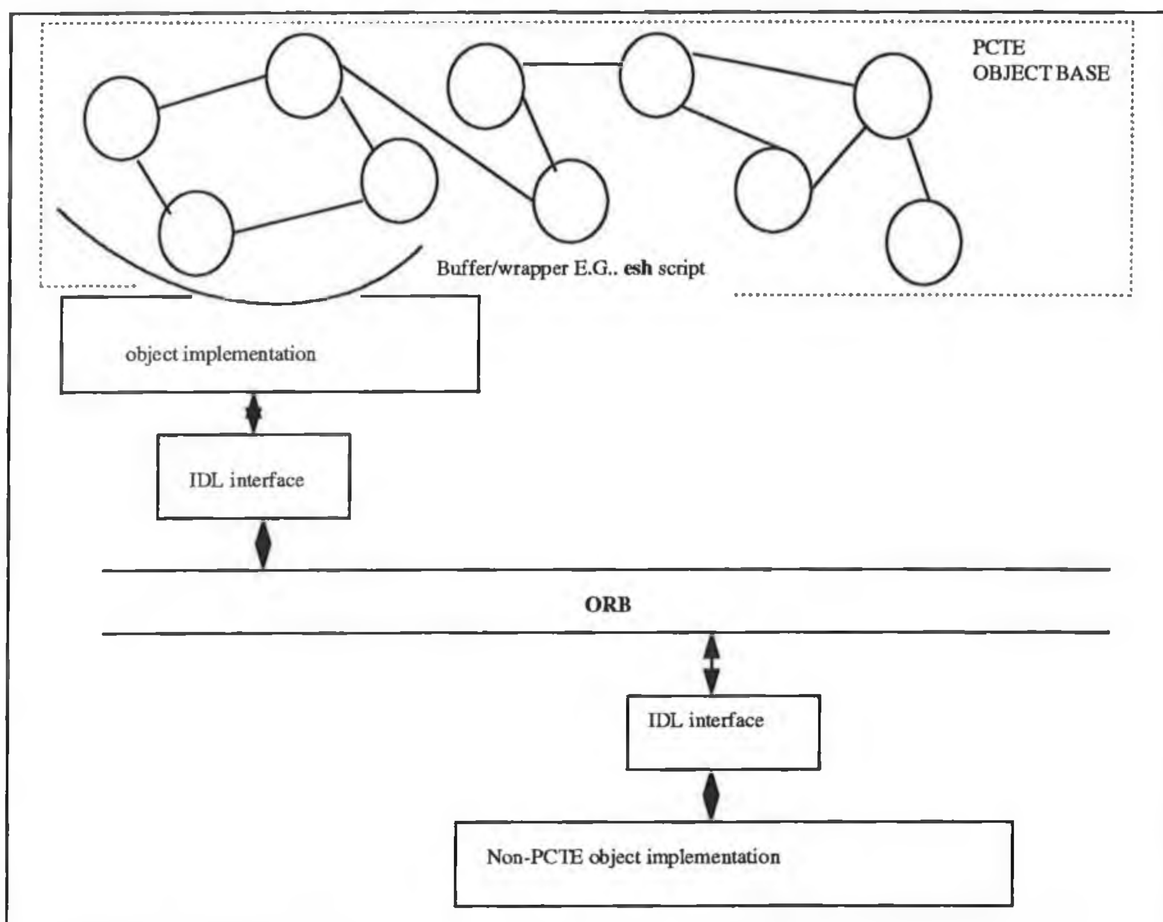

figure 6.1 *IDL interface for PCTE tools*

Therefore the object implementation of an IDL interface advertising the operations of PCTE tools must contain some method of executing the PCTE tools which are stored within the repository. The method used in this research was to embed a PCTE shell script (as opposed to a UNIX shell script) within the CORBA object implementation, see Section 6.3.1. The shell script acts as a wrapper or buffer between the CORBA object implementation and the PCTE tool. The PCTE shell script also allows us to use the PCTE activity operations (See Section 2.6) as provided by the PCTE API (Application Program Interface) to ensure that the object base remains in a consistent state.

In this way PCTE tools are wrapped in CORBA IDL interfaces, so that they can advertise their services which can be invoked by any other CORBA objects, while hiding the implementation details. This facilitates the composition of tools as described in Section 6.4.

6.2 A PCTE Tool's IDL interface

The IDL interface defined for a PCTE tool must contain an operation declaration for each service provided by the tool and the parameters that are required in order to invoke each operation. Sections 6.2 and 6.3 take a PCTE tool for editing C source code files as an example to illustrate how an IDL interface would be defined and implemented for such a PCTE tool; the full code for these examples can be found in Appendix D. The *obj_edit* PCTE tool can be used for such a purpose as long as the *c_prog* SDS is included in its working schema. The following is the IDL interface for this PCTE tool.

```
interface editor {
        readonly attribute EDITOR_RESULT changes ;
        void edit_object(in string objectname, in string e_disp);
};
```

Here the attribute *changes* is used to indicate if the file has been changed during the edit. The *edit_object* operation declaration requires two parameters to be sent to the object server (notice the **in string**), *objectname* and *e_display*. The *objectname* parameter specifies the name of the PCTE object to be edited, while *e_disp* specifies on what terminal it is to be displayed (remember PCTE is a distributed environment). Once an object implementation has been defined for this interface, by invoking *editor*'s *edit_object,* the **obj_edit** tool can be executed via CORBA to edit the C source file named as its *objectname* parameter. We will now discuss how PCTE tools can be embedded in CORBA object implementations.

6.3 Implementing a PCTE tool's IDL interface

As outlined in Section 6.1, it is the object implementation which will specify that it is a PCTE tool which will provide the services advertised in the IDL interface. This section discusses the implementation of PCTE tools' IDL interfaces and the embedding of PCTE tools in an object implementation using PCTE shell scripts. The object implementation of the IDL interface of a PCTE tool is constructed as follows. An implementation class is declared for the interface, which has a corresponding method for every operation defined in the IDL interface, and a ***set*** and ***get*** function for each attribute of the IDL interface, unless it is a **readonly** attribute, in which case only a ***get*** function is required (e.g. the

changes method defined below is the implementation of the *editor* interfaces *changes* attribute).

Take for example the extract below taken from the declaration of the implementation class, *Editor_i*, for the *editor* IDL interface, see Appendix D. The fact that the class *Editor_i* inherits from the class *editorBOAImpl* indicates that it is the implementation class for the *editor* IDL interface, notice also that the **Environment &** parameter indicates that this method is an implementation of an operation defined in the IDL interface.

```
#include "editor.idl.h"
// class Editor_i, implementation class for the editor IDL interface

class Editor_i : public virtual editorBOAImpl {
protected:
        .....
        EDITOR_RESULT changes_i ;
        .....
public :
        .....
        //      calls the edit esh wrapper to edit the PCTE C source file object
        virtual void edit_object(char *objectname, char *e_disp, Environment &);
        //      returns value of changes_i, value depending on the file being edited has
        //      changed.
        virtual EDITOR_RESULT changes(Environment &);
};
```

Note: *EDITOR_RESULT* is an enumerated IDL type defined in Appendix D.

Now that we have declared the object implementation class of the IDL interface for a PCTE tool, we must embed the PCTE tool in the methods declared by this class to be implementations of the operations in the IDL interface. This is done using PCTE shell scripts in a UNIX environment. similar facilities exist in other environments- for example in DOS, .bat files could be used. Thus a PCTE shell script is used to invoke the tool from the method. In the example given below, a further extract from Appendix D, the *Editor_i* method *edit_object* is the implementation of the operation *edit_object* defined in the *editor* IDL interface. It has embedded in it an execution of the **esh** shell script *edit*, which handles the editing of the PCTE C source file objects.

```
void Editor_i::edit_object(char *objectname,char *e_disp, Environment &)
{
        .......
        if      (pid    =       fork()) {       // fork a process to execute the script
                wait(&status);                  // Parent process waits for completion
                .......
        }
        else{
        execlp("/home/cse/emerpcte/bin/tools/environ.tools/esh","esh","edit",(char *)0);
}
```

A shell is a command interpreter that provides a user interface to a particular software environment; several shells are available to run on UNIX systems. These UNIX shells all provide command processing facilities [56] but these will not necessarily be able to access the PCTE object base. The Emeraude shell **esh** is specifically designed for exploring and modifying the object base. The shell's command interpreter has a number of facilities for generating or constructing complex commands and to write *scripts*; Section 6.3.1 describes **esh** shell scripts in greater detail.

6.3.1 Esh Scripts

An **esh** shell script is an object (in the PCTE repository) containing a set of commands that can be executed by entering the object's path name. Each shell script is an interpretable static context, where the interpreter is the shell (see Section 2.5). Scripts are a convenient way of storing a set of commands to be run more than once. The commands are put in an object of type **sctx** (static context), and can subsequently be executed in a child shell process by typing the path name of the object[55].

Emeraude **esh** scripts are similar to UNIX shell scripts as described in [56]. The following is the contents of the static context (called editor.tool) containing a script which activates the obj_edit tool:

act_start TR
obj_edit $1
act_end

The **$1** following the **obj_edit** indicates that the parameter telling **obj_edit** which PCTE object to edit will be received as a parameter to the script. The fact that the PCTE process or tool (in this example **obj_edit**) is managed by a transaction activity means that the PCTE facilitates for concurrency and integrity control are utilised, and ensures the repository is never left in an inconsistent state (see Section 2.6).

In the above example of the editor IDL interface, the method defined for *edit_object* in the object implementation calls an **esh** script which subsequently executes the editor.tool

script given above. An extract from the edit **esh** script demonstrating this is shown below, the script is given fully in Appendix D. The edit script first adds c_prog and the pact SDS, because the editor interface which we are implementing is for c source files. The *edit_object* method implementation sets up the environment variables needed for the script, i.e. OBJECTNAME (the name of the PCTE c_source object to be edited) and BACKUPNAME (the name of a PCTE object where we can backup the source file before changes are made). Once it has checked to ensure the object name exists (not shown in the extract), it backs up the object before editing it using the editor.tool script shown above. When the editing of the file has been completed it checks to see if the file was changed and returns a value to the calling function accordingly.

```
# Shell wrapper for editing c source files
#
# required environment :
#                         OBJECTNAME {file name   = 'path/filename'}
#                         BACKUPNAME { backup name = 'path/backup.c'}
#
# Add the c_prog and pact working schema to the current working schema

ws_add_sds c_prog
ws_add_sds pact

# make a backup of file before edit begins

_/sun4.toolsets/user.tools/obj_copy $OBJECTNAME $BACKUPNAME

# edit the object
_/.users/ptangney.usr/patricia.tools/editor.tool $OBJECTNAME
```

check if edited object was updated

_/sun4.toolsets/imported.tools/cmp -s $*OBJECTNAME* $*BACKUPNAME*

OBJECT_CHANGED=**$?**

0 no change, 1 if change, 2 if error

delete the backup

_/sun4.toolsets/user.tools/link_delete $*BACKUPNAME*

exit $*OBJECT_CHANGED*

6.4 Tool Composition

This approach to the integration of PCTE and CORBA allows tool composition. As stated earlier in Chapter 1, tool composition is an approach to the creation of software by *composing* existing and new elements to form larger structures, writing a minimum amount of algorithmic code to do so, thus significantly reducing the effort required to build large software systems. For example, a complete PCTE tool for building C programs may be composed from the editor IDL interface described previously in this chapter and a compiler IDL interface developed in a similar manner. Such a "building" tool would detect changes made to a C source file during an editing session, and would then automatically re-compile the edited file, displaying any errors which occurred during compilation. The IDL interface for such a composite tool is shown below, the complete source code being included in Appendix D.

```
#include "editor.idl"
#include "compiler.idl"

interface builder    :    editor, compiler {
        void build(  in string objectname,
                     in string execname,
                     in string cparameters,
                     in string disp);
};
```

Although the compiler IDL interface described in Appendix D is implemented using PCTE C compiler tools in the same fashion as was described earlier in the chapter for the editor IDL interface, this need not necessarily be the case; PCTE tools may also be integrated with non-PCTE tools using this approach, a very useful facility, see figure 5.1.

6.5 Evaluation

The previous section outlines how this integration strategy supports tool composition, allowing new software tools to be composed from existing PCTE tools and non-PCTE tools, writing a minimum amount of algorithmic code to do so. The advantage of this is a significant reduction in the effort required to build large software systems. However the tool integration model of the tool composition, provided by this strategy, does not permit the composer to chose the granularity of the composition, i.e. no choice between bindings that are either high performance with tight coupling or lower performance with lower coupling are provided by this approach. By using such a method the binding will always be medium to large grain. This is because there is still a dependency on PCTE to provide

the security and locking on an object-by-object basis, and thus all intrinsic modelling and interpretative overheads of PCTE are still incurred. This is a drawback to the effectiveness of tool composition using this approach, because it places limits on the potential performance of such composite tools. Ideally tool composition should support fine and coarse granularity. This limitation on performance and lack of support for fine grained access to the repository is compensated by the fact that the rich semantic modelling and security provided by PCTE remains intact.

While the integration strategy described in this chapter is illustrated in terms of a UNIX environment implementation of PCTE, it is not restricted to such an environment, minor adjustments making it portable to other environments, for example to DOS. This strategy is beneficial to the existing PCTE specification because it increases the amount of control integration within a PCTE environment, by allowing the co-ordination of PCTE tools via their IDL interfaces and CORBA, so enabling a closer integration between tools in a PCTE based SEE. In effect the definition of IDL interfaces for PCTE tools allows these tools to become truly object oriented because they encapsulate both the tool (behaviour) and the data objects (working schema) that the tool requires, whereas PCTE objects on their own do not model behaviour. PCTE tools which have IDL interfaces defined for them are OMA compliant; they can avail of the services of the ORB and other OMA compliant systems. And so this strategy also allows PCTE tools to be integrated with tools outside the PCTE repository.

Although the integration of this strategy is not at a fundamental level and offers no benefits to the CORBA specification, importantly, it requires no alterations to be made to either of the existing PCTE and CORBA specifications and so it can be used with the existing specifications immediately. The integration strategy described in this chapter can not be considered a mutually beneficial integration to both PCTE and CORBA. While it

offers increased control integration to PCTE, supports the composition of PCTE tools and makes PCTE fully object oriented, it does not offer the persistent storage of OMA objects originally envisaged as the benefit to CORBA of an integration between it and PCTE. However, because it offers so much to the current specification, it is a worthwhile interim integration strategy.

CHAPTER 7 CONCLUSIONS

This chapter begins by restating the objectives of the research contained in this thesis before concluding by evaluating how and with what success these objectives were achieved. Essentially, the objective for this research was that it would provide a short term mutually beneficial integration for the PCTE and CORBA specifications, without changing either of the current specifications (as described in Chapters 2 and 3 respectively), to be used while waiting for their eventual convergence. The benefits sought by such an integration were that:

- PCTE tools would be "wrapped" in an IDL interface which would allow CORBA to provide increased control integration between PCTE tools, and to make PCTE objects fully object oriented.

- The PCTE object base could be used in turn by CORBA as a persistent store for OMA objects, by availing of the rich data modelling provided by PCTE's DDL.

The reason that such emphasis was placed on the objective of using the unaltered current specifications of PCTE and CORBA was to avoid overlapping with the work of the OMG PCTE SIG, the main concern of which is to converge the two standards. So the usefulness of the research contained in this thesis is short term, for the benefits it can provide to the current specifications. Sections 7.1- 7.2 describe more fully the objectives of the integration from the PCTE and CORBA view points respectively. In Section 7.3 we discuss how and with what success these objectives were achieved.

7.1 PCTE

PCTE has become very successful as a standard for a Public Tool Interface (for an open repository) for integrated SEEs. In my opinion, this is evident from the diversity of platforms for which PCTE implementations are available, and the international support shown for the PCTE specification by its acceptance as an ISO standard in July 1994. PCTE's main strength lies in its support for data integration (with very limited control integration) and the portability of CASE tools. The importance of PCTE lies in its use as a leading specification for an open standard for integrating tools into SEEs, because it has become evident that such an open standard for integrating tools is vital to the realisation of the full potential of CASE.

PCTE has a strong object oriented flavour, the PCTE repository (which can be distributed) being composed of data objects with links showing the relationships between objects. However it is not object oriented in the truest, since it lacks a vital object oriented mechanism which would allow operations or methods to be associated with PCTE's purely data objects. Thus it was with a view to enhancing PCTE to object orientation in its purest form, and to extending the integration between PCTE tools(which is primarily based on data integration) to include a tighter control integration of a PCTE environment, that this thesis set out to integrate it (PCTE) with the OMA specifications, in particular the CORBA specification.

7.2 CORBA

The OMA specifications are defined by OMG as an infrastructure for distributed computing. They are designed to ease the development of integrated software systems across possibly heterogeneous platforms. The criterion agreed by OMG for the specification of the Object Management Architecture included the support of modular software production; that the specifications must encourage reuse of code; allow useful integration across lines of developers, operating systems and hardware; and enhance the long-range maintenance of that code. The object oriented approach to software construction was seen as the best match to this criteria, and so all the OMA specifications are to be based on this approach.

Independently developed applications which adhere to the OMA specification can be combined seamlessly in user specific ways. This is the beauty of OMA: it reduces the complexity of distributed systems. The CORBA specification forms the communication heart of the OMA specifications, and is central to the integration of distributed software systems by providing a “software bus” by which distributed OMA objects can communicate. The Interface Definition Language (IDL) has a central role to play within CORBA in order to facilitate integration. CORBA is evidently an ideal integration technology to introduce into a PCTE environment in order to increase the weak control integration or co-ordination between PCTE tools. By incorporating CORBA into the PCTE environment, CORBA (in particular IDL) can be used to associate behaviour with PCTE data objects, thus making them object oriented in the full sense.

The purpose of the OMA specifications is to “drive the industry towards interoperable, reusable, portable software components based on standard object-oriented interfaces"

[29]. However, of the OMA specifications described in Section 3.2, only CORBA is fully specified and has implementations available at the moment. The full specifications for the Object Services and Common Facilities will become available in due course, in my opinion, because of the growing popularity of distributed systems, and the importance of compliance with standards such as OMA in order to integrate such distributed systems. PCTE could be used to implement at least part of the Object Services specification, in that the PCTE repository with its rich semantic modelling could be used to provide persistent storage for OMA objects. For this reason it would be advantageous to OMA/CORBA to integrate it with the PCTE specification.

7.3 INTEGRATION STRATEGIES

Having reviewed why an integration of CORBA and PCTE is desirable, this section evaluates how successful the integration strategies explored were. In the initial stages of the search for an integration strategy, the mapping of PCTE's Data Definition Language (DDL) to CORBA's Interface Definition Language (IDL), seemed like an obvious approach, since a direct language mapping between them would allow an automatic translation of PCTE objects into CORBA objects and vice versa. This would have been ideal, as a translation tool could have been built to translate between the two languages. By mapping from IDL to DDL, CORBA/OMA objects could have been given a DDL representation and the PCTE repository utilised as a persistent store for them. The mapping of DDL to IDL would have facilitated the definition of PCTE objects as CORBA objects, able to avail of the ORB for communication and control integration as well as the other facilities provided by the OMA, including the benefits of object orientation (e.g. code reuse, ease of maintenance).

Therefore, initially, it seemed that this language mapping would satisfy all the criteria set for a mutually beneficial integration of the two specifications, including no alteration to either specification. However further research found that this was not the case for reasons that will be reiterated in Sections 7.3.1 and 7.3.2, and so a different approach was sought. In this second approach, described in Chapter 6, the definition of IDL interfaces for PCTE tools, it was obvious from the outset that the objective of using the PCTE object base as a persistent store for CORBA objects would be sacrificed, and so this approach was not going to provide a mutually beneficial integration of the two specifications. Such a sacrifice was accepted in the hope of still attaining the goal of increased object orientation and control integration between PCTE objects, Section 7.3.4 evaluates the extent to which these goals were attained.

7.3.1 DDL TO IDL

In chapter 5 we saw that, even though DDL language constructs can be mapped into IDL language constructs, the resulting interfaces are meaningless. This is because the objects defined by DDL (i.e. PCTE objects) have no behaviour. Therefore, when these objects are mapped onto IDL interfaces, the corresponding interface has no operations. The purpose of defining an IDL interface for an object is to advertise the operations or methods offered by that object to the rest of the environment. Therefore if none of the IDL interfaces which result from a mapping from DDL have any operations defined for them, the purpose of defining an IDL interface is defeated. In order to make DDL compatible with IDL, it would be necessary to extend DDL to facilitate the association of behaviour with PCTE objects. Much of the current work of the PCTE SIG is concerned with extending DDL for this purpose. Until such extensions are made to DDL, this thesis concludes the mapping of DDL to IDL is pointless as an integration strategy.

7.3.2 IDL TO DDL

As stated previously, initially, the language mapping of IDL to DDL was seen as an approach which allow OMA objects to be defined and exist in the PCTE repository, thus using the PCTE object base as a persistent store for OMA objects by using the PCTE's OMS to provide the persistent service which will form a part of future OMA Object Services specifications. From very early on, the mapping of IDL to DDL proved unfeasible for two very important reasons, PCTE objects are not compatible with the OMG Object Model, mainly because there is no mechanism for associating PCTE objects with tools, i.e. no behaviour is associated with PCTE objects. The second reason found for this incompatibility is that IDL scoping rules are incompatible with DDL scoping rules, because IDL syntax allows for the nested declaration of interfaces. In contrast, the PCTE SDSs defined using DDL are linear in nature, and therefore unable to model the possibly nested IDL interfaces and modules. Therefore, similar to the reverse language mapping, the mapping of IDL to DDL can not be achieved without extending DDL.

7.3.3 IDL interfaces for PCTE Tools

Once research proved that the language mapping of DDL to IDL (and vice versa) was not going to be successful, it was obvious that some of the objectives of the integration would not be met. Clearly the definition of IDL interfaces for PCTE tools, as described in Chapter 6, was not going to facilitate the use of the PCTE repository by CORBA as a persistent store. However it was decided that, if successful, the benefits that it would provide to PCTE environments meant that it was worth pursuing, and so the goal of a mutually beneficial integration (of PCTE and CORBA) was compromised. Instead of

working on extensions to DDL which would have paralleled the work being done by the OMG PCTE SIG, an alternative route was taken.

The definition of IDL interfaces for PCTE tools did have some success. It increased the amount of control integration within a PCTE environment, by allowing the co-ordination of PCTE tools via their IDL interfaces and CORBA, which in turn facilitated tool composition. In effect the IDL interfaces for PCTE tools allow these tools to become truly object oriented because they encapsulate both the tool (behaviour) and the data objects (working schema) that the tool requires. PCTE objects on their own do not model behaviour. This strategy importantly required no alterations to be made to either of the existing PCTE and CORBA specifications, and it also allowed PCTE tools to be integrated with tools outside the PCTE repository.

Because this integration strategy is on a superficial level, it does not provide support for fine-grained access to the repository, thus placing limits on the overall potential performance. However the rich semantic modelling and security provided by PCTE remains intact. Improved performance and support for fine-grained access to the PCTE repository is a primary consideration of the work described in [57] currently being carried out by the OMG PCTE SIG.

7.4 Future Work

The OMG PCTE SIG is currently working on a proposal for PCTE OO extensions which will integrate CORBA into the PCTE specification. The purpose of this proposal is to provide support for fine-grained access to the PCTE repository, make PCTE fully object oriented (methods will then be associated with PCTE objects), increase control integration

or co-ordination between PCTE tools, and make PCTE objects OMA compliant, therefore allowing them access to other OMA services. The areas of the PCTE specification extended (see [49]) and described below include: PCTE data modelling, the execution model and method activation, context objects, scoping of operation requests, implementation registration and server selection.

- *PCTE data modelling* The PCTE metabase and object base is extended to accommodate Object Oriented services by extending the DDL syntax (see Section 5.4), the metasds, the system SDS and providing additional SDSs.

- *Execution Model and Methods Activation* The PCTE process model is extended to support the use of the OO execution model within PCTE.

- *Context Objects* In order to stay close to the CORBA model for invocation, the extended PCTE must support context information for the dispatching of operations. This context is intended in a broad sense to include both PCTE dynamic context and CORBA invocation context [49].

- *Scoping of operation requests* The method mapping model will be exploited to scope an operation request in order to obtain different results according to the specific user/invocation context or working schema.

- *Implementation registration and server selection* An object implementation, once started, registers with the underlying run-time system, which in turn might exploit this information to select an already available implementation and/or share servers between several users.

The work in this area is ongoing: the extension outlined above will be fully specified in the future. We can see that the OMG PCTE SIG are committed to developing a more powerful PCTE specification which will be integrated with OMA/CORBA. Once these extensions have been made, particularly to the data modelling component of the PCTE specification, PCTE can then in turn be of benefit to the OMA. PCTE OMS, as pointed out earlier, provides semantically rich data modelling, because the object base or

repository is required to store and manage very complex data and relationships across the whole software life cycle- not only finished products of the software process (e.g. designs, functional specifications, alpha, beta and full tested versions of code, fault reports, change requests) but also the intermediary and supporting data that accumulates along the way (e.g. project history, test results, memos and reports) [6]. Despite the fact that PCTE was originally designed for CASE environments, many of the concepts that it has developed can be utilised for different environments. PCTE OMS's network of objects and links allows complex relationships to be modelled in an intuitive way. Future work on the Object Services component of the OMA could utilise the extended OO PCTE's OMS to provide basic operations for the logical modelling and physical storage of objects, since the extended PCTE specification would not be limited by performance as it will provide support for small grain/high speed access to the repository.

7.5 OVERALL CONCLUSIONS

The OMG PCTE SIG will provide a mutually beneficial merging of both PCTE and CORBA standards sometime in the near future. Even when such a merging has been specified, it will take even more time before an implementation is available. In the meantime the integration strategy described for the definition of IDL interfaces for PCTE tools can be used to increase the control integration between PCTE tools in a PCTE-based SEE. By using such IDL interfaces to access PCTE tools, to all clients, the tool seems truly object oriented because the object implementation of the interface encompasses both the tool's data and the static context (executing tool), its behaviour, while support for composite tools is also provided (combinations of PCTE tools and non-PCTE tools are possible). PCTE tools which are wrapped in an IDL interface can avail of the ORB and other OMA services, including other OMA compliant systems. Therefore this thesis has

illustrated that there is much is to be gained by integrating the current PCTE specification with CORBA prior to their convergence.

The mapping of DDL to IDL (and vice versa) would have provided a more beneficial and fundamental integration of both PCTE and CORBA. However this thesis has proven that such a mapping is not possible without altering the current specifications.

BIBLIOGRAPHY

[1] "*Standard ECMA-149 Portable Common Tool Environment Abstract Specification*", European Computer Manufacturers Association, 2nd Edition, June 1993.

[2] "*Standard ECMA-149 Portable Common Tool Environment Abstract Specification*", European Computer Manufacturers Association, December 1990.

[3] "*Software Engineering Economics*", Boehm B W., Prentice Hall, Englewood Cliffs, 1981.

[4] "*A Spiral Model of Software Development and Enhancement*", Boehm B W., ACM SIGSOFT Software Engineering Notes, Volume 11, 1986.

[5] "*Relationship of PCTE to OMA*", OMG TC Document 93.4.7, April 1993.

[6] "*PCTE The Standard for Open Repositories*", Lois Wakeman & Jonathan Jowett, PIMB Association, 1993.

[7] "*News*", PCTE Newsletter, No.17, PIMB, May 1994.

[8] "*News*", PCTE Newsletter, No.16, PIMB, April 1994.

[9] "*SCM Vs CASE Frameworks and repositories*", Gene Forte, CASE Outlook, Vol. 7 No. 2, 1993.

[10] "*Introduction to COHESIONworX 2.0 Linking Development Teams*", Digital, Byte Magazine, July 1994.

[11] "*COHESIONworX/PCTE: a Framework for PCTE Environments*", Augusto Argento, Chiara Bonferini, Fabrizio Dematte, Proceedings of the PCTE'94 Conference in San Francisco, USA, PIMB, 1994.

[12] "*Analyzing A Persistent Object Definition Language*", Ariela Stern, Arizona State University, May 1994.

[13] "*Developing & Integrating Tools In Eclipse/PCTE*", Sean P. MacRoibeaird, Dublin City University, May 1990.

[14] "*PCTE Functional Specifications 1.4*", Bull, GEC, ICL, Nixdorf, Olivetti, Siemens, September 1986.

[15] "*The Entity-Relationship Model: Towards a Unified View of Data*", Chen P. P., ACM Trans. on Database Systems, Vol. 1, No. 1, 1976.

[16] "*PCIS Object Oriented Services*", Timothy E. Lindquist, Proceedings of PCTE '93 Conference in Paris, published by Syntagma Systems Literature on behalf of the PIMB Association1993.

[17] "*The Object Management System of PCTE as a Software Engineering Database Management System*", Gallo, Ferdinando, Regis Minot and Ian Thomas, SIGPLAN Notices, Vol. 22, No. 1, 1987.

[18] "*Semantic Database Modelling : Survey, Application and Research Issues*", Richard Hull & Roger King, ACM Computing Survey, Vol. 19, No. 3, 1987.

[19] "*Managing the evolution of the data schemas of a PCTE-based Software Engineering Environment*", John Cheesman, Ian Simmonds (SFGL), Proceedings of the PCTE'93 Conference in Paris, published by Syntagma Systems on behalf of the PIMB Association1993.

[20] "*The Toaster Model*", Tatge G., 1989 cited in [6].

[21] "*Reference Model for Frameworks of Software Engineering Environments*", European Computer Manufacturers Association, Technical Report ECMA TR/55, 2nd Edition, December 1991.

[22] "*A fully conformant ECMA PCTE Implementation*", Jean-Claude Grosselin, Gerard Boudier, Proceedings of the PCTE'93 Conference in Paris published by Syntagma Systems Literature on behalf of the PIMB Association1993.

[23] "*Semantic Data Models*", Joan Peckham and Fred Maryanski, ACM Computing Surveys, Vol. 20, No. 3, 1988.

[24] "*PCIS Technical Study 4 -Architectural Diagrams*", Minot R, Bremeau C., PCIS/TS/S4, October 1991.

[25] "*Working Together To Integrate CASE*", Ronald J. Norman, Minden Chen, IEEE Software, March 1992.

[26] "*Definitions of Tool Integration for Environments*", Ian Thomas, Brian A. Nejmeh, IEEE Software, March 1992.

[27] "*The Future for Open Standards in CASE*", Richard Baker, CASE Outlook, Vol. 6 No. 2, March-April 1992.

[28] "*PCTE Interfaces : Supporting Tools in Software-Engineering Environments*", Ian Thomas , IEEE Software, November 1989.

[29] "*Object Management Architecture Guide*", Second Edition, OMG TC Document 92.11.1, Richard Mark Soley (ed.), OMG, September 1992.

[30] "*The Common Object Request Broker: Architecture and Specification*", Revision 1.1, OMG Document Number 91.12.1, OMG and X/Open, 1991.

[31] "*CORBA QUANDRY: Finding the Elusive Common Distributed Object*", David S. Linthicum, Application Development Trends Vol. 1 No. 11, October 1994.

[32] "*Distributed Architecture is Mission of OMG*", Brad Kain, Application Development Trends Vol. 1 No. 8, August 1994.

[33] "*Make Way for Data*". Paul Koreniowski, BYTE Vol. 18 No. 7, June 1993.

[34] "*Looking to Object Standards*", Chris Stone, Information Week New York, February 1994.

[35] "*Common Object Request Broker 2.0 And Component Object Model Interoperability Request For Proposals*", OMG TC Draft Document 94.8.31, 1994.

[36] "*OLE to Gain Object Role*", PC WEEK Medford Mass., March 1994.

[37] "*Programming in the OMG Environment*", Jon Siegel, RS/Magazine, March 1994.

[38] "*Microsoft's View : How OLE Fits*", Gregory Leake, Applications development Trends, Vol. 1 No. 11, October 1994.

[39] "*Distributed Systems Management*", Alwyn Langsford, Jonathan D. Moffett, Data Communications and Networks Series, Addison-Wesley, July 1992.

[40] "*Unravelling the Standards*", Dana M. Marks, T. Moriarty, Database Programming and Design. December 1993, Miller Freeman Publications.

[41] "*The Object Database Standard : ODMG - 93*", R. G. G. Cattell, Tom Atwood, Joshua Duhl. Guy Ferran, Mary Loomis, Drew Wade, Morgan Kaufmann Publishers 1994.

[42] "*Object Management Group : OMG forms common facilities task force & fast track adoption process. Forms Portable Common Tool Environment SIG*", EDGE : Work-Group Computing Report, January 1994.

[43] "*An ECMA PCTE Compliant Implementation Of CORBA Adding Control Facilities To ECMA PCTE Environments*", Augusto Argento, Chiara Bonferini, Fabrizio Dematte, Serena Manca (Digital Equipment Corporation, Varese, Italy), Proceedings of the PCTE' 93 Conference in Paris, published by Syntagma Systems on behalf of the PIMB Association1993.

[44] "*OOTIS Extending PCTE With Fine-Grained Tool Composition*", William Harrison, Harold Ossher, Mansour Kavianpour, PCTE Newsletter No. 11, December 1992.

[45] "*Portable Common Interface Set (PCIS) Architecture: Framework Abstract Specification*", Version 1.0, Tri-Service Group on Communications and Electronics, Special Working Group on Ada Programming Support Environments, December 1993.

[46] *"Portable Common Interface Set (PCIS) Architecture: Framework Definition and Rational"*, Version 1.0, Tri-Service Group on Communications and Electronics, Special Working Group on Ada Programming Support Environments, December 1993.

[47] *"PCIS and the Evolution of PCTE"*, M. F. Boyer, Ada Yearbook C. Loftus (Ed.) Amsterdam 1994.

[48] *"IRAC: International Requirements and Design Criteria for the Portable Common Interface Set (PCIS)"* , Version 1.0, Tri-Service Group on Communication and Electronics, Special Working Group on Ada Programming Support Environments, May 1992.

[49] *"OO (Object Oriented) Extensions to the PCTE Standard (ISO/IEC 13719)"*, Draft Version 3.0, Intended future publication of ECMA and OMG PCTE SIG, March 1995.

[50] *"DEC ACA Services Reference Manual"*, Digital Equipment Corporation, April 1992.

[51] *"ECMA PCTE, CORBA and AIS"*, A. Argento, C. Bonferini, F. Dematté, S. Manca, PCTE Newsletter No. 10.

[52] *"DEC ObjectBroker 2.5 User Guide"*, Digital

[53] *"Object Oriented Tool Integration Services (OOTIS) OOTIS Integration Model -IBM AIX-CASE proposal"*, Willam Harrison, Harold Ossher, Mansour Kavianpour, Working Draft Version, June 1992.

[54] *"PCTE SDSs for Modelling OOTIS Control Integration"*, Willam Harrison, Harold Ossher, Mansour Kavianpour, Eric Wong, Proceeding of the PCTE'93 Conference in Paris, published by Syntagma Systems Literature, on behalf of the PIMB Association1993.

[55] Emeraude V12.3.1 Documentation, GIE Emeraude.

[56] *"The UNIX Programming Environment"*, Brian W. Kernighan, Rob Pike, Prentice-Hall Software Series, 1984.

[57] *"FG (Fine Grain Data) Extensions to the PCTE Standard (ECMA-149 ISO/IEC -13719)"*, Draft Version 2.0, intended joint publication of ECMA and OMG PCTE SIG, March 1995.

[58] *"Not Your Fathers RPC"*, Jonathan Chinitz, SunExpert, Vol. 5, No.6, June 1994.

APPENDIX A Interface Definition Language (IDL)

The following clauses define the EBNF for CORBA's Interface Definition Language:

(1) <specification> ::=

<definition>+

(2) <definition> ::=

<type_dcl> ";" |

<const_dcl> ";" |

<except_dcl> ";" |

<interface> ";" |

<module> ";"

(3) <module> ::=

"**module**" <identifier> "{" <definition>+ "}"

(4) <inheritance> ::=

<nterface_dcl> |

<forward_dcl>

(5) <interface_dcl> ::=

<interface_header> "{" <interface_body> "}"

(6) <forward_dcl> ::=

"**interface**" <identifier>

(7) <interface_header> ::=

"**interface**" [<inheritance_spec>]

(8) <interface_body> ::=

<export>*

(9) <export> ::=

<type_dcl> ";" |

<const_dcl> ";" |

<except_dcl> "; |

<attr_dcl> ";" |

<op_dcl> ";"

(10) <inheritance_spec> ::=

":" <scoped_name> { "," <scoped_name> }*

(11) <scoped_name> ::=

<identifier> |

"::" <identifier> |

<scoped_name> "::" <identifier>

(12) <const_dcl> ::=

"const" <const_type> <identifier> "=" <const_exp>

(13) <const_type> ::=

<integer_tye> |

<char_type> |

<boolean_type> |

<floating_pt_type> |

<string_type> |

<scoped_name>

(14) <const_expr> ::=

<or_expr>

(15) <or_expr> ::=

<xor_expr> |

<or_expr> |

<xor_expr>

(16) <xor_expr> ::=

<and_expr> |

<xor_expr> "^" <and_expr>

(17) <and_expr> ::=

<shift_expr> |

<and_expr> "&" <shift_expr>

(18) <shift_expr> ::=

<add_expr> |

<shift_expr> ">>" <add_expr> |

<shift_expr> "<<" <add_expr>

(19) <add_expr> ::=

<mult_expr> |

<add_expr> "+" <mult_expr> |

<add_expr> "-" <mult_expr>

(20) <mult_expr> ::=

<unary_expr> |

<mult_expr> "*" <unary_expr> |

<mult_expr> "/" <unary_expr> |

<mult_expr> "%" <unary_expr>

(21) <unary_expr> ::=

<unary_operator> <primary_expr> |

<primary_expr>

(22) <unary_operator> ::=

"-" |

"+" |

"~"

(23) <primary_expr> ::=

<scoped_name> |

<literal> |

"(" <const_expr> ")"

(24) <literal> :=

<integer_literal> |

<string_literal> |

<character_literal> |

<floating_pt_literal> |

<boolean_literal>

(25) <boolean_literal> ::=

"**TRUE**" | "**FALSE**"

(26) <positive_int_const> ::=

<const_exp>

(27) <type_dcl> ::=

"**typedef**" <type_declarator> |

<struct_type> |

<union_type> |

<enum_type>

(28) <type_declarator> ::=

<type_spec> <declarators>

(29) <type_spec> ::=

<simple_type_spec> |

<constr_type_spec>

(30) <simple_type_spec> ::=

<base_type_spec> |

<template_type_spec> |

<scoped_name>

(31) <base_type_spec> ::=

<floating_pt_type> |

<integer_type> |

<char_type> |

<boolean_type> |

<octet_type> |

<any_type>

(32) <template_type_spec> ::=

<sequence_type>

| <string_type>

(33) <constr_type_spec> ::=

<struct_type> |

<union_type> |

<enum_type>

(34) <declarators> ::=

<declarator> { "," <declarator> }*

(35) <declarator> ::=

<simple_declarator> |

<complex_declarator>

(36) <simple_declarator> ::=
<identifier>

(37) <complex_declarator> ::=
<array_declarator>

(38) <floating_pt_type> ::=
"**float**" |
"**double**"

(39) <integer_type> ::=
<signed_int>
| <unsigned_int>

(40) <signed_int> ::=
<signed_long_int> |
<signed_short_int>

(41) <signed_long_int> ::= "**long**"

(42) <signed_short_int> ::= "**short**"

(43) <unsigned_int> ::=
<unsigned_long_int> |
<unsigned_short_int>

(44) <unsigned_long_int ::= "**unsigned**" "**long**"

(45) <unsigned_short_int> ::= "**unsigned**" "**short**"

(46) <char_type> ::= "**char**"

(47) <boolean_type> ::= "**boolean**"

(48) <octet_type> ::= "**octet**"

(49) <any_type> ::= "**any**"

(50) <struct_type> ::=
"**struct**" <identifier> "{" <member_list> "}"

(51) <member_list> ::= <member>+

(52) <member> :: =
<type_spec><declarators> "**;**"

(53) <union_type> ::=
"**union**" <identifier> "**switch**" "(" <switch_type_spec> ")"

(54) <switch_type_spec> ::=
<integer_type> |
<char_type> |
<boolean_type> |
<enum_type> |
<scoped_name>

(55) <switch_body> ::= <case>+

(56) <case> ::=
<case_label> + <element_spec> "**;**"

(57) <case_label> ::=
"**case**" <const_exp>"**:**" |
"**default**" "**:**"

(58) <element_spec> ::=

<type_spec> <declarator>

(59) <enum_type> ::=

"**enum**" <identifier> "{" <enumerator> {"," <enumerator> } *}

(60) <enumerator> ::= <identifier>

(61) <sequence_type> ::=

"**sequence**" "<" <simple_type_spec> "," <positive_int_const> ">" |

"**sequence**" "<" <simple_type_spec> ">"

(62) <string_type> ::=

"**string**" "<" <postive_int_cosnt> ">"

(63) <array_declarator> ::=

<identifier> <fixed_array_size>+

(64) <fixed_array_size> ::=

"[" <positive_int_const> "]"

(65) <attr_dcl> ::=

["**readonly**"] "**attribute**" <simple_type_spec> <declarators>

(66) <except_dcl> ::=

"**exception**" <identifier> "{" <member>* "}"

(67) <op_dcl> ::=

[<op_attribute>] <op_type_spec> <identifier> <parameter_dcls>

[<raises_expr>] [<context_expr>]

(68) <op_attribute> ::= "**oneway**"

(69) <op_type_spec> ::=

<simple_type_spec> |

"**void**"

(70) <parameter_dcls> ::=

"(" <param_dcl> {"," <param_dcl> }* ")"

(71) <param_dcl> ::=

<param_attribute> <simple_type_spec> <declarator> |

"(" ")"

(72) <param_attribute> ::=

"**in**" |

"**out**" |

"**inout**"

(73) <raises_expr ::=

"**raises**" "(" <scoped_name> { "," <scoped_name> }* ")"

(74) <context_expr> ::=

"**context**" "(" <string_literal> { "," <string_literal>}* ")"

Appendix B Data Definition Language (DDL)

This appendix contains the EBNF of PCTE's Data Definition Language.

(1) DDL definition =

sds section, {sds section};

(2) sds section =

'**sds**', sds name, ';',

{clause, ';'}

'**end**', sds name, ';';

(3) clause =

type importation | object type declaration |

object type extension | attribute type declaration |

link type declaration | link type extension |

enumeration type declaration;

(4) type importation =

'**import**', import type, global name,['**as**', local name],[type mode declaration], {',', global name, ['**as**', local name],[type mode declaration]};

(5) import type =

'**object**', '**type**' | '**attribute**', '**type**' |

\`**link**\`, \`**type**\`;

(6) object type declaration =

local name, ';',[type mode declaration], [`**child**`,`**type**`,`**of**`, object type list], [`**with**', ['**contents**', contents type indication, ":"],
['**attribute**', attribute indication list, ";"],
['**component**', component indication list,";"]
'**end**', local name];

(7) object type extension =

'**extend**', '**object**', '**type**', local name, '**with**',
['**attribute**' indication list,";"],
['**link**', link indication list, ';']
['**component**', component indication list, ';']
'**end**', local name;

(8) contents type indicatio =

'**file**' | '**pipe**' | '**device**' | '**audit_file**' | '**accounting_log**';

(9) attribute indication list=

attribute indication list item {';', attribute indication list item};

(10) attribute indication list item =

attribute type name | attribute type declaration;

(11) link indication list =

link indication list item {';', link indication list item}

(12) link indication list item =

link type name | link type declaration ;

(13) component indication list =

component indication list item, { ';', component indication list item}

(14) component indication list =

link type name | link type declaration;

(15) attribute type declaration =

local name, {',' local name}, ':', [type mode declaration],['**non_duplicated**'], value type indication, [':=', initial value];

(16) value type indication =

'**integer**' | '**natural**' | '**boolean**' |
'**time**' | '**float**' | '**string**' |
'**enumeration**', enumeration type name |
enumeration type indication;

(17) enumeration type indication =

'**enumeration**', '(', basic enumeration, {',', basic enumeration}, ')';

(18) basic enumeration =

enumeration image | enumeration subrange ;

(19) enumeration image =

identifier | '"', {character}, '"';

(20) enumeration subrange =

attribute type name, '**range**', enumeration imange, '..', enumeration image;

(21) initial value =

['+' | '-'], digit, {digit} (* Integer *)

| digit, {digit} (* Natural *)

| **'true'** | **'false'** (* Boolean *)

| year, '-', month, '-', day, ['T' hour, ':', minute, ':', second], 'Z' (* Time *)

| ['+' | '-'], digit, {digit}, ['.', digit, {digit}], ['E', ['+' | '-'], digit, {digit}] (* Float *)

| '"', {character}, '"'

| enumeration image;

(22) day =

digit, digit;

(23) month =

digit, digit;

(24) year =

[digit, digit] digit, digit;

(25) hour =

digit, digit ;

(26) minute =

digit, digit;

(27) second =

digit, digit;

(28) link type declaration =

local name, ':', [type mode declaration], ['**exclusive**'], ['**non_duplicated**'], [stability name], category name, '**link**', [cardinality range], [key list], ['**to**', object type list], ['**reverse**', link type name], ['**with**',

'**attribute**',

attribute indication list, ';',

'**end**', local name];

(29) link type extension =

'**extend**', '**link**', '**type**', local name, ['**to**', object type list], ['**with**',

'**attribute**'

attribute indication list, ';',

'**end**', local name];

(30) category name =

['**composition**'] | '**existence**' | '**reference** | '**implicit**' | '**designation**';

(31) cardinality range =

'[', [lower bound],'..',[upper bound], ']';

(32) lower bound =

digit, {digit};

(33) upper bound =

digit, {digit};

(34) stability name =

'**atomic**', '**stable**' | '**composite**', '**stable**';

(35) key list =

'(' attribute indication list, ')';

(36) enumeration type declaration =

local name, ':', enumeration image, {',' enumeration image };

(37) type mode declaration =

'(' 'usage', type mode, ';', 'export', type mode, ')' |

'(', ['usage', ',', 'export'], type mode, ')';

(38) type mode =

'protected' | allowed access, { ',', allowed access };

(39) allowed access =

'read' | **'write'** | **'navigate'** | **'create'** |

'delete';

(40) object type name =

global name | local name;

(41) object type list =

object type name, { ',', object type name};

(42) attribute type name =

global name | local name;

(43) attribute type list =

attribute type name, { ',', attribute type name};

(44) link type name =

global name | local name ;

(45) link type list =

link type name, { ',' , link type name }

(46) enumeration type name =

global name | local name ;

(47) sds name =

identifier;

(48) local name =

identifier;

(49) global name =

sds name , '-', local name;

(50) identifier =

letter, { letter | digit | '_'};

(51) capital letter =

'A'	|	'B'	|	'C'	|	'D'	|	'E'	|
'F'	|	'G'	|	'H'	|	'I'	|	'J'	|
'K'	|	'L'	|	'M'	|	'N'	|	'O'	|
'P'	|	'Q'	|	'R'	|	'S'	|	'T'	|
'U'	|	'V'	|	'W'	|	'X'	|	'Y'	|
'Z';									

(52) small letter =

'a'	|	**'b'**	|	**'c'**	|	**'d'**	|	**'e'**	|
'f'	|	**'g'**	|	**'h'**	|	**'i'**	|	**'j'**	|
'k'	|	**'l'**	|	**'m'**	|	**'n'**	|	**'o'**	|
'p'	|	**'q'**	|	**'r'**	|	**'s'**	|	**'t'**	|
'u'	|	**'v'**	|	**'w'**	|	**'x'**	|	**'y'**	|
'z';									

(53) letter =

capital letter | small letter ;

(54) digit =

'0'	|	**'1'**	|	**'2'**	|	**'3'**	|	**'4'**	|
'5'	|	**'6'**	|	**'7'**	|	**'8'**	|	**'9'**;	

(55) comment =

'--', [character], newline;

Appendix C c_prog SDS

This section describes the c_prog SDS which is used by programming tools within a PCTE environment. It contains the following type definitions:

Object types	**archive_file**
	asm_source
	c_source
	dir
	evolution
	file
	group
	include_file
	include_library
	lint_library
	object
	object_code
	program
	project
	sctx
	subset
	subset_interface
	test
	testset
	toolset
	user
Attribute types	**cause**
	edition

	name
	nature
	number
	passed
	release
	subname
	system
	system_release
	target
	variant
	version
Link and relationship types	**a**
	acts
	build
	c
	debug
	deliverable
	derived_from <-> derived_in
	e
	err
	exec
	h
	i
	inc
	include <-> included_in
	interface
	l
	ln
	modif
	monitor

o

out

output

product

prog

s

sub

subprog

testform

testin

testout

testref

tests

theme

tmp

tool

tst

v

y

The following gives a brief description of the purpose of each of these object, attribute, relationship and link types.

Object Types

archive_file This object type represents an archive file.

asm_source This object type represents an assembler source file.

c_source This object type represents a file containing C language compilable source code.

dir This object type represents a directory.

evolution This object type represents a deliverable that has evolved from a stable deliverable.

file This object type represents a temporary file, an error file, a test file, an output file, a debug file, an activities file or a **yacc** text file.

group This object type represents a group of users.

include_file This object type represents a C include file.

include_library This object type represents a library of include files.

lint_library This object type represents a lint library

object This object type is the common ancestor type of all other object types.

object_code This object type represents a file containing object code.

program This object type represents a piece of software, and has been imported from the pact SDS.

project This object type represents a software development project.

sctx This object type represents a static context.

subset This object type represents a subset of a program.

subset_interface This object type represents a description of a module's interface and will normally be used for documentation purposes.

test This object type represents a software test.

testset This object type represents a set of tests.

toolset This object type represents a collection of static contexts.

user This object type represents a user of the Emeraude environment.

Attribute Types

cause This attribute type indicates the reason for an evolutionary derivation of software.

edition This attribute type indicates the edition number of a piece of software.

name This attribute type indicates the name of an object. It is typically used as the key on a link to the object.

nature This attribute type represents a short description of the sort of change involved in an evolution, what is being tested and a summary of a test set.

number This attribute type is used to distinguish between instances of the same link type originating from the same object.

passed This attribute type indicates whether or not a piece of software has passed a quality test.

release This attribute type indicates the release number of a piece of software.

subname This attribute type indicates a secondary name of an object. It is typically used with name as a key on a link to the object.

system This attribute type indicates the name of the system under which the software development is taking place.

system_release This attribute type indicates the release number of the system under which the software development is taking place.

target This attribute type indicates the hardware on which the developed software is designed to execute.

variant This attribute type indicates the variant name of a piece of software.

version This attribute type indicates the version number of a piece of software.

Link and Relationship Types

a This is a link to an archive file.

acts This is a link from a subset to a temporary file created and used by the Unix yacc tool.

build This is a link from a piece of software to a collection of tools.

c This is a link from a subset to a C compilable source file.

debug This is a link from a subset to a debug file, created and used by the Unix yacc tool.

deliverable This is a link from a piece of software to a stable object representing a deliverable program.

derived_from <-> derived_in This is a relationship between two objects, one being a derivation of the other.

e This link type is provided to allow compatibility with Unix file systems.

err This is a link from a subset to an object representing an error file.

exec This is a link from a piece of software to a static context that it requires to execute.

h This is a link from an include library or a subset to an include file.

i This is link from a subset to a C source file, to hold output from the C pre-processor.

inc This is a link from a piece of software to an include library.

include <-> included_in This is a relationship between a stable include file and a C compilable source file or another include file.

interface This is a link from a subset to its interface file (that holds a description of the module subset).

l This is a link from a subset to a temporary file, created and used by the Unix tool **lex**.

ln This is a link from a directory to a link library.

modif This is a link from any object to an evolution, representing a modification to the software development.

monitor This is a link from a test input file to the static context representing a test monitor.

o This is a link from a directory or a subset to a file containing object code. It enables binaries to be collected for any purpose.

out This is a link from a piece of software to a static context. It is the C compiler default.

output This is a link from a subset to an output file, created and used by the Unix tool yacc.

product This is a link from a software development project to a piece of software.

prog This is a link from a user or group to a piece of software.

s This is a link from a subset to an assembler source file.

sub This is a link from a piece of software to a subset or from one subset to another.

subprog This is a link from one piece of software to another, and represents the association between the two.

testform This is a link from a test to a file which holds a complete description of a test. The attribute type nature represents a brief description only.

testin This is a link from a test to a file containing commands to initialise a debugging session.

testout This is a link from a test to a file containing the output of a test session.

testref This is a link from a test to a file containing reference output.

tests This is a link from a program or subset to the set of tests to be applied to that program or subset.

theme This is a link between two sets of tests and represents the association between the two sets of tests.

tmp This is a link from a subset to a temporary file.

tool This is a link from a collection of tools to a static context.

tst This is a link from a set of tests to a test belonging to the set.

v This is a link from a user or a group to a piece of software.

y This is a link from a subset to an input text file fro the Unix compiler yacc.

DDL listing for the c_prog SDS

The following is the DDL listing for the c_prog SDS taken from the public types of [58]:

```
new_sds     c_prog is

import      sys-name    as      name ;

release         :       integer         :=      1;

version         :       integer         :=      1;

edition         :       integer         :=      1;

system          :       string ;

system_release          :       string ;

target          :       string  ;

variant         :       string  ;

number          :       integer         :=      1;

passed          :       boolean  ;

subname         :       string  ;

nature          :       string  ;
```

```
cause            :  string :=  "bug" ;

import  sys-object    as  object  ;
import  sys-file      as  file    ;
import  sys-dir       as  dir     ;
import  sys-sctx      as  sctx    ;
import  env-group     as  group   ;
import  env-project   as  project ;
import  env-user      as  user    ;
import  env-toolset   as  toolset ;
import  pact-software as  program ;

include_library  :  subtype  of  object ;
include_file     :  subtype  of  file   ;
c_source         :  subtype  of  file   ;
asm_source       :  subtype  of  file   ;
object_code      :  subtype  of  file   ;
archive_file     :  subtype  of  file   ;
```

```
lint_library      :  subtype      of     file    ;

test              :  subtype      of     object  ;

testset           :  subtype      of     object  ;

subset            :  subtype      of     object  ;

subset_interface  :  subtype      of     file    ;

evolution         :  subtype      of     file    ;

prog              :  composition  link   ( name )
                     to   program        ;
```

```
import       env-tool     as     tool    ;

extend       tool    to    sctx    ;

import       env-e        as     e       ;

extend       e       to    include_library      ;

tst               :    composition  link   ( name, number )
                       to     test    ;

h                 :    composition  link   ( name, subname )
                       to     include_file    ;

product           :    composition  link   ( name, release )
                       to     stable  program       ;

deliverable       :    reference    link   ( number )
                       to     stable  object  ;

sub               :    composition  link   ( name)
                       to     subset  ;

inc               :    composition  link   ( name )
                       to     include_library      ;

build             :    composition  link   to     toolset ;

modif             :    composition  link   ( number )
                       to     evolution     ;
```

```
relationship  (
        derived_from  :     reference     link
        to      object  ;
        derived_in    :     implicit      link
        to      object  )   ;

c                     :     composition  link   ( name, subname )
                            to     c_source     ;

tests                 :     composition  link   to    testset ;

monitor               :     composition  link   to    sctx    ;

testin                :     composition  link   to    file    ;

testref               :     composition  link   to    file    ;

testout               :     composition  link   to    file    ;

exec                  :     composition  link   to    sctx    ;

interface             :     composition  link   to    subset_inteface     ;

i                     :     composition  link   ( name, subname )
                            to     c_source     ;

a                     :     composition  link   ( name, subname )
                            to     asm_source   ;

err                   :     composition  link   ( name, subname )
                            to     file    ;
```

```
relationship  (
        include       :   reference     link   (name)
                          to   stable include_file   ;
        included_in   :   implicit             link   ()
                          to   c_source,   include_file   )   ;

y                     :   composition   link   ( name, subname )
                          to   file   ;

output                :   composition   link   ( name, subname )
                          to   file   ;

debug                 :   composition   link   ( name, subname )
                          to   file   ;

acts                  :   composition   link   ( name, subname )
                          to   file   ;

tmp                   :   composition   link   ( name, subname )
                          to   file   ;

l                     :   composition   link   ( name, subname )
                          to   file   ;

ln                    :   composition   link   ( name, subname )
                          to   lint_library   ;

subprog               :   composition   link   ( name, subname )
                          to   program   ;

testform              :   composition   link   to   file   ;
```

```
theme          :    composition  link   ( name )
                    to     testset ;

o              :    composition  link   ( name, subname )
                    to     object_code   ;

v              :    composition  link   ( name,version )
                    to     program       ;

out            :    composition  link   ( name, subname )
                    to     sctx    ;

extend object
        with
        link    modif  ;
end     object ;

extend dir
        with
        link    ln     ;
                a      ;
                e      ;
                o      ;
end     dir    ;

extend group
        with
        link    prog   ;
                v      ;
end     group  ;
```

```
extend project
    with
    link    product     ;
end    project;

extend program
    with
    attribute    version:
                 edition :
                 system :
                 system_release      ;
                 target :
                 variant:
    link         deliverable    ;
                 sub    :
                 inc    :
                 build  :
                 tests  :
                 exec   :
                 subprog     ;
                 a      ;
                 out    ;
end    program      ;

extend user
    with
    link    prog    ;
            v       ;
end    user    ;
```

```
extend toolset
        with
        link    tool    ;
end     toolset ;

extend include_library
        with
        link    h       ;
                e       ;
end     include_library ;

extend test
        with
        attribute       passed ;
                        nature ;
        link            monitor ;
                        testin ;
                        testref ;
                        testout ;
                        testform ;
end     test ;

extend testset
        with
        attribute       nature ;
        link            tst    ;
                        theme  :
end     testset ;
```

```
extend subset
    with
    link    h        ;
            sub      ;
            interface    ;
            c        ;
            i        ;
            a        ;
            s        ;
            err      ;
            y        ;
            output   ;
            debug    ;
            acts     ;
            tmp      ;
            l        ;
            o        ;
            tests    ;
end     subset ;

extend evolution
    with
    attribute    cause  ;
                 nature ;
end     evolution

end     c_prog
```

Appendix D Example IDL interface for PCTE tools

In this example, an IDL interface is defined for two PCTE tools, an Editor (obj_edit) and a C compiler (ecc). This example also demonstrates tool composition: via CORBA and their IDL interfaces, these tools were combined to form a builder tool, which edits a C source code file, and if any changes are made to the file during the editing session the file is recompiled automatically. This appendix contains the complete souce code for this example. ORBIX Version 1.1 from IONA Technology is the CORBA implementation and Emeraude PCTE V 12 is the PCTE implementation used in this example.

The following is the IDL definition for the interface to the PCTE compiler, **ecc**, compiler.idl :

```
enum COMPILER_RESULT {
        COMPILER_FAILED,          //compiler could not be executed
        COMPILED_OK,              //compiled with no erors
        COMPILED_WITH_ERRORS,     //compiled with errors
        COMPILER_NOT_TRIED }; //compiler not invoked

interface   compiler {
        readonly attribute COMPILER_RESULT errors;

        void compile( in string objectname,
                      in string execname,
                      in string parameters,
                      in string disp);
};
```

The following is the IDL interface definition for the PCTE tool editor, **obj_edit**, *editor.idl* :

```
enum EDITOR_RESULT {
                    EDIT_FAILED,
                    FILE_CHANGED,
                    FILE_UNCHANGED};

interface    editor {
             readonly attribute EDITOR_RESULT changes ;
             void edit_object(    in string objectname,
                                  in string e_disp);
};
```

The following is the IDL interface definition for the composite tool, *builder.idl*:

```
#include "editor.idl"
#include "compiler.idl"

interface builder: editor, compiler {
        void build(   in string objectname,
                      in string execname,
                      in string cparameters,
                      in string disp);
};
```

The class *Editor_i* is the implementation class for the ***editor*** IDL interface. The class declaration and definition for *Editor_i* is given below.

```
//      file name : editor_i.h
#include "editor.idl.h"

//      class Editor_i
//      Editor_i interfaces to the Edit wrapper

class   Editor_i : public virtual editorBOAImpl {
protected:
        char e_display[150];            // holds the screen display
        char pathname[150];
        EDITOR_RESULT changes_i ;
        void set_changes(EDITOR_RESULT number);
private:
        char * object_path(char *);
public:
        Editor_i();
        virtual void edit_object(       char *objectname,
                                        char *e_disp,
                                        Environment &);
        virtual EDITOR_RESULT changes(Environment &);
};
```

```
//        file name : editor_i.cc

#include <iostream.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "editor_i.h"
#include "demo.h"

// function definitions for the Editor_i class

extern char **environ;                // holds the environment variables used be execlp

Editor_i::Editor_i()
{
        changes_i=0 ;           // initialise changes
        strcpy(pathname,""); //intialise pathname
}

char * Editor_i::object_path(char *objectname)
{
        // truncate the file name from the end of the object name to
        // reveal the path name

        int     i       =       0 ;
        int     slen    =       0 ;
        char    obj_name[150];
        char    path[150];

        strcpy(obj_name,objectname);
```

```
        strcpy(path,"");
        slen    =       strlen(objectname);
        for     (i=slen;            obj_name[i]!='/';       i--);
        strncpy(path,objectname,i);
        strcat(path,"/backup.c");
        return(path);
}

void    Editor_i::edit_object( char *objectname,
                               char *e_disp,
                               Environment &)
{
        int     status ;
        int     pid ;
        char    env_string[150] ;
        char * dummy ;

//      set up the environment in which the child process is to execute

        strcpy(e_display,e_disp);
        environ[0] = new char[150];
        strcpy(env_string,"FILENAME=");   // set up FILENAME environment var
        strcat(env_string, objectname);
        strcpy(environ[0],env_string);

        environ[1] = new char[150];
        strcpy(env_string,"DISPLAY=");      // set up DISPLAY environment variable
        strcat(env_string,e_display);
        strcpy(environ[1],env_string);
        dummy=object_path(objectname);

        environ[2] = new char[150];
```

```
strcpy (env_string,"BACKUPNAME=");      // set up DISPLAY variable
strcat(env_string, dummy);
strcpy(environ[2],env_string);

if    (    pid    =fork())    {    // Parent has non zero [True] pid
        wait(&status); // Parent process waits for completion
        switch(status) {
                case   CLEAN_VAL_1_RETURN:
                                set_changes(FILE_CHANGED);
                                break;
                case   CLEAN_VAL_0_RETURN:
                                set_changes(FILE_UNCHANGED);
                                break;
                default:        set_changes(EDIT_FAILED);
                                break;}
}
else                    // Child has zero [FALSE] pid
{
        execlp( "/home/cse/emerpcte/bin/tools/environ.tools/esh",
        "esh","edit",
        (char *)0);  // child
        }
}

void Editor_i::set_changes(EDITOR_RESULT number)
{
//      set changes to reflect if the file has been changed

        changes_i    =    number  ;
}
```

```
EDITOR_RESULT Editor_i::changes(Environment &)
{
// returns value of changes_i, value depends on the file being edited has changed.
        return(changes_i);
}
```

The following is the ESH script wrapper which interfaces between the PCTE tool obj_edit, and the implementation of the **editor** interface.

```
# Shell wrapper for PCTE edit tool
# required environment :
#                       FILENAME {file name   = 'path/filename'}
#                       BACKUPNAME { backup name = 'path/backup.c'}
#
# Shell type : ESH

# Add the c_prog and pact working schema to the current working schema

ws_add_sds c_prog
ws_add_sds pact

# set the home object

co ~ _/.users/ptangney.usr

# check to see if object exists

els $OBJECTNAME | grep -s "1.."
case $? in
        0)      ;;
```

```
        2)      STATUS = 3; exit $STATUS;; # system error
        1)      _/.users/ptangney.usr/patricia.tools/pcteOC.tool $OBJECTNAME
                # pcte tool to create an object of type c_source
                case $? in
                0);;       # object created ok, zero bytes long
                *)      echo "not created error";
                        exit 2;; # object not created due to error
                esac
;;
esac


# make a backup of file before edit begins

 _/sun4.toolsets/user.tools/obj_copy  $OBJECTNAME     $BACKUPNAME

# edit the object

 _/.users/ptangney.usr/patricia.tools/editor.tool      $OBJECTNAME


# check if edited object was updated

 _/sun4.toolsets/imported.tools/cmp -s       $OBJECTNAME     $BACKUPNAME

# 0 no change, 1 if change, 2 if error
OBJECT_CHANGED=$?

# delete the backup

_/sun4.toolsets/user.tools/link_delete $BACKUPNAME
exit $OBJECT_CHANGED
```

The class *compiler_i* is the implementation class for the ***compiler*** IDL interface. The class declaration and definition for *compiler_i* is given below.

```
//      compiler_i.h

#include "compiler.idl.h"

class compiler_i : public virtual compilerBOAImpl
{
    protected:
        COMPILER_RESULT error_i; //error status of last invocation
            //set error to result of last compile
        void set_error(COMPILER_RESULT result) {error_i = result;}
        char c_display[150]; // holds screen display
    public:
        //constructor creates object and inits compiler error status
        compiler_i() {error_i = COMPILER_NOT_TRIED;}
        //get last compile result
        virtual COMPILER_RESULT errors(Environment &)
                                {return(error_i);}
        virtual void compile( char *objectname,
                              char *execname,
                              char *parameters,
                              char *disp,
                              Environment &);
    };
```

```
//      file name : compiler_i.cc

#include <iostream.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "compiler_i.h"
#include "demo.h"

extern char **environ        ; //environment info in this data struct used by "execlp"

void compiler_i::compile(    char *filename,
                             char *execname,
                             char * parameters,
                              char *disp,
                             Environment &)
{
        int     status  ;               // Status of child at termination
        int     pid     ;               // pid = 0 for child, non-zero for parent
        char    env_string[150];        // Unix environment values for child.

        //      set up unix environment for child processes
        environ[0]      =       new char[150];
        strcpy(environ[0],"OBJECTNAME=");
        strcat(environ[0],filename);
        environ[1]      =       new char[150];
        strcpy(environ[1],"PARAMS=");
        strcat(environ[1], parameters);

        environ[2]      =       new char[150];
```

```
strcpy(env_string,"DISPLAY=");    // set up DISPLAY env var
strcat(env_string,disp);
strcpy(environ[2],env_string);

environ[3]    =    new char[150];
strcpy(env_string,"EXECNAME=");
strcat(env_string,execname);
strcpy(environ[3],env_string);

if    (    pid    =    fork())        // Parent has non zero [True] pid
{
        wait(&status); // Parent process waits for completion
        switch(status)
        {
                case    CLEAN_VAL_0_RETURN:
                                set_error(COMPILED_OK);
                                break;

                case    CLEAN_VAL_1_RETURN:
                                set_error(COMPILED_WITH_ERRORS);
                                break;

                default:        set_error(COMPILER_FAILED);
                                break;
        }
}
else                    // Child has zero [FALSE] pid
        {
execlp(         "/home/cse/emerpcte/bin/tools/environ.tools/esh",
                "esh",
                "compile",(char *)0);          /* child */
        }       }
```

The following is the ESH script wrapper which interfaces between the PCTE tool **ecc,** and the implementation of the **compiler** interface.

```
# Required environment : OBJECTNAME, DISPLAY, EXECNAME
# Shell type : ESH

# Check if object exists if not then exit with error
# set up PCTE working schema
ws_add_sds c_prog
ws_add_sds pact

els $OBJECTNAME | grep -s "1.."
case $? in
  0) ;; # object exists
  *) STATUS=3; exit $STATUS ;; # system error
 esac

# Check if the execution object has been created

els $EXECNAME | grep -s "1.."
case $? in
        0) ;; #object exists
        *) _/.users/ptangney.usr/patricia.tools/execOC.tool $EXECNAME
        case $? in
                0);;    # created ok
                *)      echo "ERROR: Unable to create execution object !";
                        STATUS=3; exit $STATUS;;
        esac ;;
esac

# Compile object with PCTE tools
```

```
# Check what parameters have been passed to the wrapper
        P="";

        echo $PARAMS | grep -s "NOLINK"
        case $? in
                1) P="-c";;
                *) ;;
        esac

        echo $PARAMS | grep -s "OPTIMISE"
        case $? in
                1) P="-O";;
                *) ;;
        esac

        echo $PARAMS | grep -s "DEBUG"
        case $? in
            1) P="-g";;
            *) ;;
        esac


 ./sun4.toolsets/imported.tools/ecc  $P  -o  $EXECNAME  $OBJECTNAME   2>
errorlist;
      case $? in
           0) STATUS=0; echo "compled ok." ;;
           1) STATUS=1; echo "errors during compilation";;
       esac

cat errorlist
exit $STATUS;
```

The class *builder_i* is the implementation class for the ***builder*** IDL interface. The class declaration and definition for *builder_i* is given below.

```
#include "builder.idl.h"
#include "editor_i.h"
#include "compiler_i.h"

// class builder_i

class builder_i : public virtual builderBOAImpl, public virtual Editor_i, public
virtual compiler_i {
public :
        builder_i(); // constructor
        virtual void build(     char *objectname,
                                char *execname,
                                char *parameters,
                                char *display,
                                Environment &);
};


//      builder_i.cc


#include <iostream.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "builder_i.h"
```

```
#include "demo.h"


//        function definitions for the builder_i class

builder_i::builder_i() { }


void builder_i::build( char *objectname,
                       char *execname,
                       char *parameters,
                       char *display,
                       Environment &)
{

        compiler *    CompilerPtr;
        editor   *    EditorPtr;
        int           number_changes;
        int           number_errors;

        TRY {
                CompilerPtr = compiler::_bind("","sonia",IT_X);
        } CATCHANY {
                cout << "\n Error in binding to a compiler interface " << IT_X;
                exit(1);
        } ENDTRY

        TRY   {
                EditorPtr = editor::_bind("","sonia",IT_X) ;
        } CATCHANY {
                cout << "\n Error in binding to an editor interface " << IT_X;
                exit(1);
```

```
} ENDTRY

TRY {
        EditorPtr->edit_object(objectname,display,IT_X);
 } CATCHANY {
        cout << "\n Error invoking the Editor " << IT_X;
        exit(1);
 } ENDTRY

TRY {
 number_changes = EditorPtr->changes(IT_X);
} CATCHANY {
        cout << "\n Error detecting Changes \n" << IT_X;
        exit(1);
} ENDTRY

if      ( number_changes == FILE_CHANGED ) {

TRY {
CompilerPtr->compile(objectname,execname,parameters,display,IT_X);
   } CATCHANY {
     cout << "\n Error invoking Compiler \n << IT_X";
     exit(1);
   } ENDTRY
TRY {
     number_errors = CompilerPtr->errors(IT_X);
   } CATCHANY {
     cout << "\n Error detecting compile errors \n "<< IT_X;
     exit(1);
   } ENDTRY
```