Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware

Fábio Moreira Costa

B.Sc., Federal University of Goiás, Brazil, 1992
M.Sc., State University of Campinas, Brazil, 1995

Computing Department
Lancaster University

A thesis submitted for the degree of Doctor of Philosophy

August 2001
Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware

Fábio Moreira Costa

B.Sc. (Federal University of Goiás, Brazil, 1992)
M.Sc. (State University of Campinas, Brazil, 1995)

Department of Computer Science,
Lancaster University,
Lancaster, LA1 4YR

Submitted for the degree of Doctor of Philosophy,
August 2001

Abstract

The last decade has seen the emergence and popularisation of middleware technologies, such as the OMG’s Common Object Request Broker Architecture (CORBA), Microsoft’s Distributed Component Object Model (DCOM) and Sun’s Java RMI. These technologies have profoundly changed the way distributed applications are developed. In particular, they enable the large-scale deployment of applications in distributed heterogeneous environments, by making transparent the distribution aspects and enabling the developers to focus on application concerns.

Importantly, the dissemination of distributed systems technology has also stimulated the emergence of new application areas, such as distributed multimedia and mobile applications. However, due to the dynamic nature of these new applications, current middleware technologies have proven inadequate. The major reason for this is the currently dominating “black-box” philosophy for middleware design, which precludes the flexible configuration and adaptation of a platform in order to suit the particular and evolving requirements of different applications.

This thesis proposes a solution to these limitations by adopting an open-ended approach to the design of middleware platforms. To this end, a middleware architecture is developed which combines the use of meta-information management techniques, for the definition of customised platform configurations, with object-oriented reflection, which allows the dynamic adaptation of the platform. The integration of the two techniques is based on a well-defined meta-model, which prescribes, in a unified way, the structure and contents of the meta-information handled by both the meta-information management facility and the reflection
mechanisms. This combined approach facilitates a consistent view of the different phases of a platform’s lifecycle, as the same meta-information is used during design, deployment and runtime. The approach also enables the use of reflection for type evolution, in order to dynamically generate new platform configurations, which can be stored in a repository for reuse. In addition, this use of reflection can be constrained by type evolution rules, so that inconsistent reconfigurations can be avoided. A prototype implementation of this architecture is described, allowing the demonstration and evaluation of the feasibility and the properties of the approach.
Declaration

I declare that the work presented in this thesis is my own and has not been submitted in this or any form for a research degree. The work has been developed within the context of the Open-ORB Reflective Middleware project at the Computing Department, Lancaster University, which provided the basis for the architecture proposed in this thesis. The extensions and refinements to the Open-ORB approach that were presented in this thesis, notably the integration with meta-information management, have been entirely carried out by the author, as well as all work related to the design and implementation of the proposed architecture. An exception to this is the part related to local bindings, which has been re-used from the OOPP platform, developed by Anders Andersen, of the University of Tromsø, Norway.

This research was jointly funded by the Computing Department at Lancaster University, by The Brazilian Council for Scientific and Technological Development, and by the Federal University of Goiás.
Acknowledgements

I would like to thank everyone within the Distributed Multimedia Research Group at Lancaster University for providing a constructive environment for the development of the research presented in this thesis. In particular, I want to thank Professor Gordon Blair, for his endless patience and encouragement, and for his invaluable advice and feedback throughout the development of this research. Working through this Ph.D. under his supervision was one of the most enriching experiences that I’ve had. I also want to thank Geoff Coulson, Katia Saikoski, Nikos Parlavantzas, Rui Moreira, Hector Duran, Anders Andersen, Tom Fitzpatrick, Lee Johnston, Mike Clarke and Paul Grace, for their helpful advice and discussion. A big thank you goes to Matt Storey, for his great help with equipment, especially the laboratory used during the evaluation work.

I must also thank The Brazilian Council for Scientific and Technological Development (CNPq) and the Federal University of Goiás (UFG), for providing the financial support that made this work possible. Particularly, the Institute of Informatics at UFG must be thanked for allowing me the necessary time to dedicate to the achievement of this degree.

On a more personal note, I want to give special thanks to my darling wife for her unwavering love and inestimable support, and to my little son, Felipe, for all the joy and happiness he has given us. I love to you both! I also want to thank my parents, sisters and also my dear relatives and friends, both in Brazil and here in Lancaster, who kept providing important encouragement.

And finally, I want to express my thankfulness to Jesus, for He is before all things, and by Him all things consist. Thank you Lord!

August 2001
To Ester
Table of Contents

CHAPTER 1 INTRODUCTION.............................................................................................................. 1
  1.1 OVERVIEW.......................................................................................................................... 1
  1.2 MIDDLEWARE AND OPEN DISTRIBUTED PROCESSING.................................................. 2
    1.2.1 Overall concepts, standards and technologies......................................................... 2
    1.2.2 Emerging applications and their need for flexible middleware support.................. 3
    1.2.3 Discussion .................................................................................................................... 4
  1.3 EXPLORING THE META-LEVEL......................................................................................... 5
    1.3.1 Motivation ................................................................................................................... 5
    1.3.2 Reflection and Meta-level Architectures ..................................................................... 5
    1.3.3 Meta-Information Management .................................................................................. 7
  1.4 OBJECTIVES...................................................................................................................... 9
  1.5 THESIS STRUCTURE ....................................................................................................... 10

CHAPTER 2 FUNDAMENTALS OF REFLECTIVE MIDDLEWARE............................................. 12
  2.1 INTRODUCTION ................................................................................................................ 12
  2.2 PRINCIPLES OF REFLECTION AND META-LEVEL ARCHITECTURES........................... 12
    2.2.1 Foundations .................................................................................................................. 12
    2.2.2 Styles of reflection ....................................................................................................... 14
    2.2.3 Structure of the meta-level ......................................................................................... 16
  2.3 REPRESENTATIVE REFLECTIVE ARCHITECTURES ................................................... 17
    2.3.1 Outline ......................................................................................................................... 17
    2.3.2 Reflection in programming languages ........................................................................ 17
    2.3.3 Reflection in operating systems ................................................................................ 18
    2.3.4 Distributed reflective architectures ............................................................................ 19
    2.3.5 Other areas of application ......................................................................................... 20
  2.4 REFLECTIVE MIDDLEWARE ......................................................................................... 21
    2.4.1 Overview and motivation ............................................................................................. 21
    2.4.2 Evaluation framework ................................................................................................ 22
  2.5 REFLECTION FACILITIES IN CURRENT MIDDLEWARE TECHNOLOGIES .................. 26
    2.5.1 Overview ..................................................................................................................... 26
    2.5.2 OMG CORBA .............................................................................................................. 27
    2.5.3 Microsoft COM, COM+ and .NET ............................................................................ 28
    2.5.4 RM-ODP ....................................................................................................................... 30
    2.5.5 Discussion ................................................................................................................... 31
  2.6 SURVEY OF REFLECTIVE MIDDLEWARE RESEARCH ............................................. 32
    2.6.1 Overview .................................................................................................................... 32
    2.6.2 FlexiNet ...................................................................................................................... 33
    2.6.3 OpenCORBA ............................................................................................................... 34
    2.6.4 2K, dynamicTAO and UIC™ ..................................................................................... 35
    2.6.5 DART .......................................................................................................................... 37
    2.6.6 ROBR ......................................................................................................................... 38
    2.6.7 Quarterware ............................................................................................................... 38
    2.6.8 mChaRM .................................................................................................................... 39
    2.6.9 Discussion and comparative analysis .................................................................... 39
  2.7 ALTERNATIVE AND COMPLEMENTARY TECHNIQUES ........................................... 41
  2.8 SUMMARY ....................................................................................................................... 42

CHAPTER 3 META-INFORMATION MANAGEMENT.................................................................. 44
  3.1 INTRODUCTION ................................................................................................................. 44
  3.2 CONCEPTS AND APPLICATIONS ..................................................................................... 45
    3.2.1 Meta-information ....................................................................................................... 45
    3.2.2 Related concepts ....................................................................................................... 45
    3.2.3 Management of meta-information ............................................................................. 48
3.2.4 Relationship to reflection .......................................................... 48
3.2.5 Applications ............................................................................ 50
3.3 META-INFORMATION MANAGEMENT FOR MIDDLEWARE .................. 51
3.3.1 Basic requirements ................................................................. 51
3.3.2 Type and configuration management ....................................... 52
3.3.3 Other applications in middleware ........................................... 56
3.4 KEY STANDARDS ........................................................................ 56
3.4.1 The CORBA Interface Repository ........................................... 56
3.4.2 The Meta-Object Facility and related standards ....................... 60
3.4.3 The RM-ODP Type Repository Function .................................. 67
3.4.4 Discussion ............................................................................. 69
3.5 OTHER RELATED TECHNOLOGIES AND STANDARDS .................... 71
3.5.1 The Open Information Model and the Microsoft Repository ....... 71
3.5.2 CDIF .................................................................................... 72
3.5.3 Java Meta-data API ................................................................. 72
3.6 SUMMARY ................................................................................ 73

CHAPTER 4 THE META-ORB META-MODEL ............................................ 74
4.1 INTRODUCTION ........................................................................... 74
4.2 OVERALL ORGANISATION OF THE META-MODEL ....................... 75
4.2.1 Outline ................................................................................. 75
4.2.2 Scope ................................................................................ 75
4.2.3 Modular structure ............................................................... 76
4.3 CORE INFRASTRUCTURE ............................................................. 78
4.3.1 Rationale ............................................................................ 78
4.3.2 Local bindings ................................................................. 78
4.3.3 Interface references and implicit binding ................................ 79
4.4 DETAILED DESCRIPTION OF THE META-MODEL ..................... 79
4.4.1 Basic constructs ................................................................ 79
4.4.2 Interfaces ........................................................................ 83
4.4.3 Media types ..................................................................... 87
4.4.4 QoS annotations .............................................................. 88
4.4.5 Components .................................................................... 90
4.4.6 Bindings ........................................................................ 94
4.5 SUMMARY .............................................................................. 102

CHAPTER 5 META-ORB: REFLECTION FRAMEWORK ................................ 104
5.1 INTRODUCTION ........................................................................... 104
5.2 STRUCTURING THE META-LEVEL ................................................ 104
5.2.1 Basic principles ................................................................. 104
5.2.2 Multi-model reflection framework ....................................... 106
5.2.3 Structural meta-space models ............................................. 109
5.2.4 Additional considerations .................................................. 112
5.3 REFLECTION AND META-INFORMATION MANAGEMENT: PRINCIPLES FOR AN INTEGRATED APPROACH ........................................................................ 114
5.3.1 Outline ............................................................................. 114
5.3.2 The reflection and the meta-information hierarchies .......... 115
5.3.3 Initialising the state of meta-objects .................................... 116
5.3.4 Meta-object definition ...................................................... 119
5.4 REFLECTION AND TYPE EVOLUTION ........................................ 121
5.4.1 Overall approach .............................................................. 121
5.4.2 The impact of adaptation on type evolution ....................... 123
5.4.3 Support for type versioning ................................................. 124
5.4.4 Type evolution constraints .............................................. 126
5.4.5 Major features and benefits of the approach ....................... 127
5.5 RELATED WORK ......................................................................... 128
5.6 SUMMARY ................................................................................ 129

CHAPTER 6 THE META-ORB: CONCRETE DESIGN AND IMPLEMENTATION ...... 131
6.1 INTRODUCTION ........................................................................... 131

viii
Figures

Figure 2.1 – Reification, reflection and the tower of meta-levels ......................................... 13
Figure 2.2 – Approach to separation of concerns: Middleware as part of the base-level .......... 25
Figure 3.1 – Self-definition of the CORBA IR: (a) conceptual; (b) actual .......................... 58
Figure 3.2 – Containment hierarchy of the CORBA 3.0 Interface Repository ...................... 59
Figure 3.3 – Example of use of the MOF meta-modelling architecture .............................. 62
Figure 3.4 – The MOF and its relation to the OMG architecture and to XML .................... 64
Figure 3.5 – Model and meta-model management with the MOF ..................................... 66
Figure 4.1 – Kinds of meta-information represented in the meta-model ............................... 75
Figure 4.2 – Package structure of the Meta-ORB meta-model ........................................... 77
Figure 4.3 – The BaseIDL package .................................................................................. 80
Figure 4.4 – The Interfaces package ................................................................................. 84
Figure 4.5 – The MediaDefs package ............................................................................... 87
Figure 4.6 – QoSDefs package .......................................................................................... 89
Figure 4.7 – Graphical representation for components and their interfaces ....................... 91
Figure 4.8 – The Components package ............................................................................. 91
Figure 4.9 – A composite component and its internal configuration ................................... 92
Figure 4.10 – The Bindings package ................................................................................. 96
Figure 4.11 – Graphical representation for binding objects .............................................. 97
Figure 4.12 – A multi-level composite binding .................................................................. 99
Figure 4.13 – Outline of the abstract binding protocol .................................................... 101
Figure 5.1 – The per-object relationship between base- and meta-objects ......................... 106
Figure 5.2 – Reifying a base-level object according to multiple meta-space models .......... 107
Figure 5.3 – The two hierarchies of meta ......................................................................... 115
Figure 5.4 – Joining the partial configurations of multiple binding endpoints ...................... 119
Figure 5.5 – Defining the meta-level objects in terms of the Meta-ORB meta-model ........... 120
Figure 5.6 – Impact of adaptation on the type of an object .............................................. 122
Figure 6.1 – Naming scheme for the different kinds of objects in the Meta-ORB .................. 134
Figure 6.2 – The standard component and interface types for component factories ............. 142
Figure 6.3 – Component type for local binding factories .................................................. 144
Figure 6.4 – Primary interface of local binding factories ................................................... 145
Figure 6.5 – Secondary interface of local binding factories (and accessory definitions) ......... 146
Figure 6.6 – Collect interface of local binding factories (and accessory definitions) .......... 146
Figure 6.7 – Illustration of the distributed binding protocol ............................................ 147
Figure 6.8 – Operations of the basic MOP ...................................................................... 164
Figure 6.9 – The causal-connection link .......................................................................... 166
Figure 7.1 – Component instantiation times ....................................................................... 173
Figure 7.2 – Recursive composition experiment ............................................................... 174
Figure 7.3 – Test configuration for binding objects (with two endpoints) ............................ 175
Figure 7.4 – Binding establishment time (minimal configuration) ....................................... 176
Figure 7.5 – Reifying the Architecture meta-space of composite components .................. 179
Figure 7.6 – Reifying the Architecture meta-space of binding objects ................................ 179
Figure 7.7 – End-to-end delay through pipelines of primitive components and local bindings, compared with pipelines of Python method calls (using buffers of 1024 bytes) .................. 181
Figure 7.8 – Delay for synchronous requests through a simple distributed binding ............. 183
Figure 7.9 – Estimation of the end-to-end delay for stream interactions ............................. 184
Figure 7.10 – Throughput of stream interactions, compared with UDP sockets ................. 185
Figure 7.11 – Binding configuration for the audio distribution case study ......................... 189
Figure 7.12 – Definition of the composite and primitive bindings used in the case study ....... 190
Figure 8.1 – Operational interface definition ................................................................... 241
Figure 8.2 – Primitive component definition ...................................................................... 241
Figure 8.3 – Media type system and QoS base definitions ................................................ 242
Figure 8.4 – Stream interface and media specification definitions ..................................... 243
Figure 8.5 – Composite component definition .................................................................. 244
Figure 8.6 – Binding definition ....................................................................................... 245
Tables

Table 2.1 – Summarising the features and limitations of existing reflective middleware .......................... 40
Table 3.1 – Levels of a meta-modelling architecture .............................................................................. 47
Table 3.2 – Areas where meta-modelling and meta-information management can be applied, in the
context of the representation of modelling languages ........................................................................ 51
Table 3.3 – Major constructs of the MOF Model .................................................................................... 63
Table 3.4 – Main elements of the MOF-to-IDL mapping ...................................................................... 65
Table 3.5 – Concepts defined by the ODP Type Repository (TR) Function .......................................... 69
Table 4.1 – Standard repository access operations .............................................................................. 82
Table 6.1 – Operations of the name server ............................................................................................. 135
Table 6.2 – Interface Discovery MOP (interface type: InterIDiscMeta_interf) ..................................... 158
Table 6.3 – Interface MOP (interface type: InterMeta_interf) ............................................................... 159
Table 6.4 – Architecture MOP (interface type: ArchMeta_interf) ........................................................... 159
Table 7.1 – Interface instantiation time ................................................................................................. 172
Table 7.2 – Average execution times for typical reconfiguration operations ....................................... 179
Table 7.3 – Example object graph description for the case study ....................................................... 194
Table 8.1 – Comparing the Meta-ORB reflective middleware architecture with the requirements and
design principles introduced in Chapter 2 ............................................................................................. 208
Table A.2 – Reflective middleware comparative analysis .................................................................... 213
Table A.3 – Reflective middleware comparative analysis (cont.) ......................................................... 214
Table A.4 – Reflective middleware comparative analysis (cont.) ......................................................... 215
Table A.5 – Reflective middleware comparative analysis (cont.) ......................................................... 216
Chapter 1  Introduction

1.1  Overview

Over the past decade, middleware platforms have become the dominant approach in the field of distributed systems. This has been reflected in the proposal and consolidation of standards and technologies such as RM-ODP [ITU-T/ISO 1995a], CORBA [OMG 2001a], Java RMI [Sun 2000], DCOM [Microsoft 2000c] and .NET [Microsoft 2000e]. The widespread adoption of these standards has decisively contributed to leverage the scope of distributed systems beyond the limits of homogeneous and closed environments. In addition, the high level of abstraction and transparency promoted by such standards has began to inspire the large-scale development of distributed applications in domains and environments that transcend the traditional style of client-server interaction. The most notable examples have been the emergence of distributed multimedia applications and mobile computing environments. Despite such success, however, fundamental limitations still exist in mainstream middleware technologies when it comes to fully supporting these new areas of application. One important requirement presented by such new applications refers to the need for flexibility of the underlying middleware support, which is crucial to match their highly changeable environments and user needs. The currently dominant approach of “one size fits all” of mainstream middleware technologies makes such a requirement impossible to meet.

This thesis is focused on the above requirement, aiming at a principled approach to support both short-term dynamic adaptation and long-term evolution of middleware platforms and applications. In doing so, the thesis also provides a generic framework for building customised middleware platforms. The approach is based on the combination of two distinct but complementary techniques, namely computational reflection and meta-information management. Reflection allows a computing system to perform computation about its own internal implementation, enabling dynamic inspection and change of its structure and behaviour. Meta-information management, in turn, makes practical the unified modelling and maintenance of explicit meta-information describing the configuration of a system. The combination of these two techniques enables a unified approach for the modelling and adaptation of middleware
and application configurations, in terms of a \textit{meta-model} that defines the constructs used in their constitution.

This chapter is structured as follows. Section 1.2 outlines the general area of middleware, focusing on the limitations of current technologies in face of the requirements presented by emerging categories of applications. Section 1.3 presents the two complementary techniques of reflection and meta-information management. The major concepts used in the thesis are introduced, along with the limitations of each technique, with a view towards their integration. Section 1.4 then presents a statement of the aims and objectives of the thesis, while section 1.5 concludes with a discussion of the structure of the thesis.

\section{1.2 Middleware and Open Distributed Processing}

\subsection{1.2.1 Overall concepts, standards and technologies}

In the past, programming models for distributed systems, such as RPC [Birrel and Nelson 1984], usually offered a minimum level of abstraction, typically with only the aspects purely related to communications being effectively hidden from the programmer. Consequently, many non-functional aspects related to distribution were usually programmed alongside application-specific functionality and also with direct use of the underlying operating system services. Besides the complexity of building distributed applications in this way, such approaches are likely to produce proprietary and incompatible solutions. Middleware platforms have thus been proposed with the goal of being an integrative layer of software between applications and the operating systems. This layer should offer the common distributed systems services in a way that is independent from specific applications, operating systems and hardware, thus providing an appropriate abstraction level for building distributed applications.

The overall principles for the design and development of middleware platforms have been prescribed by the ISO RM-ODP (Reference Model for Open Distributed Processing) collection of standards [ITU-T/ISO 1995d]. RM-ODP introduces a framework for building \textit{open distributed systems}, based on the provision of a comprehensive set of distribution functions and transparencies. It also introduces distinct viewpoints to facilitate separation of concerns in the design and implementation of distributed systems. In addition, the framework sets the foundation
for an object-oriented programming model for distributed systems and applications, offering an abstraction level where distributed objects can be programmed in a similar way as their centralised counterparts. Issues such as remote and heterogeneous access, location, replication, migration and interoperability are entirely managed by the platform, transparently to the applications programmer.

A number of mainstream middleware technologies realise, to some extent, the principles of RM-ODP, offering their own interpretations of the concepts presented in the standard. Arguably, the most notable technology has been the OMG Common Object Request Broker Architecture (CORBA) [OMG 2001a], which has gained widespread acceptance, and which follows a standardisation process that is closely aligned to that of RM-ODP. Notably, CORBA addresses the issues of openness, both from the perspective of programming languages and execution environments. Microsoft technologies, such as COM/DCOM [Microsoft 2000c] and .NET [Microsoft 2000e] also address the issue of openness across language boundaries, although mostly restricted to the Windows environment. In contrast, Java RMI, from Sun Microsystems [Sun 2000], enables openness across heterogeneous operating system and hardware, although limited to the Java Virtual Machine. These technologies will be considered in Chapter 2, when contrasting them with reflective middleware.

### 1.2.2 Emerging applications and their need for flexible middleware support

Despite their widespread use and adoption, current middleware standards and technologies still have significant deficiencies related to the support that they offer to emerging categories of applications and to important new requirements of traditional applications. Distributed multimedia, for example, is a relatively unexplored area of application that often requires a great degree of flexibility from the underlying middleware [Blair and Stefani 1997]. Such applications need adaptable middleware support, based on the ability to dynamically reconfigure the platform in response to changes in its environment (e.g. resource availability). Importantly, adaptation must be achieved without severely disrupting the application functionality. These requirements become even more stringent when multimedia applications (and indeed other kinds of applications) run in mobile computing environments, where changes in the environment tend to be more dramatic. Another example where flexible middleware support is required is found in the area of embedded and ubiquitous
computing environments, which have very limited resource availability. The dominant requirement here is the ability to customise middleware configurations, enabling the definition and deployment of platforms that suit the resources available in each particular environment. As a final example, consider the requirements posed by long-lived applications, which must be executed continuously and without service disruption, such as in banking systems. Despite their stability, such applications can still evolve due to new user requirements, which may lead to the need to dynamically adapt the underlying middleware (e.g. to provide new non-functional properties).

1.2.3 Discussion

Current middleware technologies only provide partial support for the above emerging application requirements. Mostly because these technologies normally mandate a rigid and closed configuration in many aspects of the platform design, it is not feasible to tailor their support in order to suit different application needs. Technologies such as Minimum CORBA [OMG 2001a] specify a compact version of the platform that is suitable for environments with limited resources. However, the standard approach is not flexible, as it prescribes a fixed configuration, which is not customisable and may not suit all cases. The same is true regarding service adaptability. For instance, both CORBA and DCOM allow the addition of new, external services to the ORB by means of interceptors that are hooked at specific points of the communications path. Nonetheless, the aspects that can be adapted through interceptors, as well as the ability to reconfigure them, are limited. Another example is the use of policy objects in CORBA, which enables several aspects of the platform, such as messaging, real-time behaviour and server adaptation (by the portable object adaptor), to be overridden in order to better suit application needs. Again, however, the installation of policies is restricted to initialisation time, thus not allowing dynamic reconfiguration. Finally, the RM-ODP framework is a step towards eliminating such limitations, as its engineering language (see Chapter 2) enables configuration of the internals of a platform. However, the level of flexibility is still coarse-grained and restricted to a few aspects that can be configured. In addition, the standard does not specify facilities for dynamic reconfiguration. A deeper analysis of the features that enable more flexible support in current mainstream middleware technologies and standards, such as those mentioned above, is provided in Chapter 2.
In summary, the main shortcomings of these technologies are due to the lack of dynamic openness regarding their internal infrastructures, which hinders the provision of services that suit applications across the different dimensions in which their requirements may vary. The thesis therefore concentrates on a middleware framework that contributes to eliminate these limitations, offering a flexible, open and principled approach for configuration and dynamic reconfiguration of platforms. The foundations of this approach are outlined in the next section.

### 1.3 Exploring the meta-level

#### 1.3.1 Motivation

In order to achieve the goals of implementation openness and dynamic adaptability, it is crucial to be able to explicitly represent and manipulate the internal implementation of a platform. Several current middleware technologies are beginning to incorporate such features, although in an *ad hoc* way (as briefly considered in section 1.2.2), meaning that only a fixed set of aspects are considered. In contrast, this thesis adopts a principled foundation, which enables the meta-representation and explicit manipulation of virtually any aspect of a platform's structure and behaviour. The dynamic manipulation of this meta-representation, with matching effects on the corresponding runtime entities of the platform, is enabled by the use of *computational reflection*. On the other hand, a unified schema for representing, defining and maintaining the required meta-information is provided by the use of techniques of *meta-information management*. These two techniques are briefly described below.

#### 1.3.2 Reflection and Meta-level Architectures

**Basic concepts and applications**

Reflective computing systems have been described as systems that are capable of reasoning about themselves, based on a principled way to explicitly represent and manipulate their implementations [Smith 1982]. This general definition permits a number of interpretations and applications of the concept. The particular interpretation adopted in this thesis is based on an explicit separation of the base- and meta-level of a system, on the fundamentals of object-oriented reflection, and on the distinction between structural and behavioural reflection.
Chapter 1 – Introduction

The base-level consists of the usual aspects of a system, which represent and manipulate concepts of the domain of application. The meta-level on the other hand, is concerned with the representation of the system itself, by reifying its internal structures and functionality. Importantly, such meta-level representation is causally connected with the base-level, meaning that changes at one level have corresponding effects at the other. In object-oriented reflection [Maes 1987], the entities that populate the meta-level are called meta-objects. These are normal objects that happen to encapsulate the meta-representation of other objects in the base-level, offering a well-defined interface through which they can be manipulated. Together, the facilities provided by the meta-object’s interface are called a meta-object protocol (MOP) [Kiczales et al. 1991]. Finally, for the purposes of this thesis, an important classification of reflective facilities divides them into structural and behavioural reflection [Malenfant et al. 1996]. Structural reflection is concerned with the reification of the functional aspects of base-level objects, such as their modular decomposition and their interfaces. Behavioural reflection, on the other hand, is concerned with the environment within which base-level objects execute, including their non-functional properties. Examples of this include the semantics of message passing, the rules for class inheritance and the mechanisms implicit in object instantiation. This thesis, however, is mainly concerned with the aspects of structural reflection that are manifest in middleware platforms for distributed systems.

The case for reflective middleware

Reflection can, in principle, be applied to any kind of execution environment where implementation openness is an important goal. In particular, it has been successfully applied as an architectural principle for programming languages, operating systems and distributed systems. Examples are described in Chapter 2. Important in the context of this thesis, however, is the use of reflection in middleware architectures. In fact, it has been advocated that middleware platforms are the most appropriate place for the comprehensive provision of reflective features [Blair and Paphthomas 1997]. This is due to an interesting synergy between reflection and middleware. On the one hand, reflection offers a principled approach to provide the dynamic flexibility that is now required from middleware platforms [Costa et al. 2000b]. To complement this, middleware provides a programming environment that masks the differences between distinct programming languages and operating systems, enabling uniform reflection
facilities (and MOPs) in heterogeneous environments. This argument in favour of reflective middleware has been confirmed by the great level of interest in the area recently [Blair and Campbell 2000], as also reviewed in Chapter 2.

Discussion

There are some limitations, however, when reflection is applied in the context of distributed systems and middleware. The most obvious one is related to the complexity of the meta-level. In a middleware scenario, the basic separation of concerns between base- and meta-level is not sufficient, as the meta-level and its interfaces (MOP) tend to be either too complex and heavyweight or otherwise limited to a rather small set of features that can be reified. Another limitation of reflection techniques in this context refers to their safety and security, as the effects of reflective computation can be spread in a distributed environment, potentially affecting several users. It is therefore necessary to provide mechanisms and policies to avoid inconsistent uses of reflection, as well as to limit potential side effects and restrict access to sensible MOP operations. Finally, the performance of the reflection mechanisms needs to be adequately managed so as not to limit the applicability of a platform.

This thesis addresses the first of the above limitations by adopting an approach similar to the multi-model reflection framework proposed in [Okamura et al. 1992]. The basic idea is to structure the meta-level in terms of distinct aspects or models, thus enabling separation of concerns at the meta-level itself. The thesis also partially addresses the second limitation by adopting a type management facility (outlined in the next section) integrated with the reflection mechanisms. This allows extensive use of type-checking and type evolution constraints in order to validate the results of reflective computation. The third limitation, however, is out of the scope of this thesis, although there is related work at Lancaster focusing on efficient implementations of reflective middleware, as described in [Clarke et al. 2001].

1.3.3 Meta-Information Management

Basic concepts and applications

Meta-information management can be generically defined as the ability to store, maintain and process, in a distributed fashion, information that describes other
information [Crawley et al. 1997a]. In the context of software systems in general, while the system represents and manipulates information about its application domain, the associated meta-information manager deals with (meta-)information that describes the system itself and the structures it uses to represent application-related information. This technique finds application in the most varied areas, such as data warehousing, workflow process management, and multimedia content processing. The growing interest in meta-information is demonstrated by the standardisation activities in the field, as surveyed in chapter 3. In particular, OMG has introduced a set of standards for the handling and management of meta-information, exemplified by the Meta-Object Facility (MOF) [OMG 2000c] and the XML Meta-data Interchange format (XMI) [OMG 2000e]. This suggests a trend towards adopting meta-information management as a central concept in middleware, which is further confirmed by the recent proposal of the Model Driven Architecture, by OMG [Soley 2000].

MIM and middleware

In the scope of this thesis, meta-information management is applied specifically to type management in middleware [Brookes and Indulska 1994], enabling first-class treatment of the types that describe the runtime entities in a middleware environment. This is made feasible as middleware platforms usually have well-defined type systems (cf. meta-models), which govern the structure and (partly) the semantics of their types. Such type systems can be seen as closed sets of meta-types (types of types) that represent the several aspects that compose type definitions. For instance, a meta-type for interfaces represents all the features that are needed to define and interpret particular interface types. In addition, meta-types also define the kinds of relationships that may exist between types, such as inheritance and substitutability.

Meta-information management can thus be explored as a basis for configuration and reconfiguration of middleware platforms. This is founded on the use of meta-types to explicitly model the structure of the constructs used in the constitution of a platform. In other words, the type system represents the language to describe middleware configurations. Crucially, this enables middleware descriptions to be manipulated as first-class entities, which in turn allows them to be used by factories in order to generate specific instances of the platform. In addition, such descriptions serve as a runtime record of the structure of a platform, which can be used to initialise
the reconfiguration mechanisms. It is important to note, however, that meta-information management can also be applied to other aspects of middleware, such as behavioural and semantic properties. This, however, will not be explored in this thesis.

Discussion

Meta-information management provides an inherent form of reflection as it can be used for the meta-representation of systems. In a type management context, for instance, types provide an explicit representation of their instances, and are first-class entities that can be manipulated at runtime. However, especially in the context of middleware, the use of this form of reflection is only practical for the purpose of inspection. This is because several instances can be derived from (and represented by) the same type, meaning that it is not possible to express reflective operations that affect just a single instance, separately from other instances of the same type. Such an approach is clearly inappropriate for middleware, where instances of the same type usually represent unrelated runtime entities, which should be able to evolve independently from each other (e.g., instances of the same interface type, but used in distinct contexts, should be treated separately from each other). In this case, the per-instance meta-level representation provided by object-oriented reflection seems more appropriate. Meta-information management, however, offers the important ability to manage meta-representations in a unified way and at a global level, rather than at the level of individual meta-objects [Costa and Blair 2000]. This suggests an important synergy between both techniques, which is of crucial importance in this thesis.

1.4 Objectives

The thesis has the overall aim of exploring architectural principles for the design and implementation of configurable and reconfigurable middleware platforms, considering the requirements posed by dynamic applications, such as those involving multimedia and mobility. More specifically, meta-information management techniques are proposed as the foundation for middleware configurability, whereas computational reflection is adopted as the basis for dynamic adaptation and reconfigurability. Importantly, the thesis investigates the relationship between the two techniques and proposes an integrated approach based on their complementary nature. The thesis therefore seeks to combine the best of both worlds, namely the intensional
Chapter 1 – Introduction

style of reflection represented by meta-information management and the more extensional style represented by object-oriented reflection. To summarise, the goals of the thesis are:

• to consider the principles of the design of middleware platforms, aiming at the aspects that are influenced by the requirements of dynamic applications;
• to examine the role of meta-information management in middleware platforms, investigating the use of meta-information modelling architectures in this context;
• to investigate the application of object-oriented reflection in the context of middleware, exploring its use to open up the platform implementation and offer the necessary adaptation support required by dynamic applications; and
• to investigate the complementary nature of the above two techniques, exploring their integration to unify the meta-representation of middleware used during design-time configuration and runtime reconfiguration.

The resulting research will be materialised by the design and prototyping of a proof-of-concept platform, as a means to show how the above concepts fit together in a middleware framework. The work aims to demonstrate the feasibility and the benefits of an integrated approach combining meta-information management and reflection. The intent, however, is to emphasise the completeness and usability of the programming model, rather than its performance-related aspects (although, as seen in Chapter 7, a comparative study of such aspects, with respect to representative baseline platforms, is important to demonstrate the relative performance of the approach).

Importantly, the thesis will mainly concentrate on the structural aspects of middleware, such as architectural composition and interface definitions. Behavioural aspects, although amenable to the modelling and manipulation through meta-information management and reflection, are the subject for related and future work.

1.5 Thesis structure

The next two chapters present the background for the work. In Chapter 2, the concept and the basic principles of reflective architectures are presented, followed by their application in the context of middleware platforms. The chapter also defines a framework of requirements for reflective middleware and examines related work in the area. Chapter 3, in turn, provides a survey of the several approaches and related
work in the area of meta-information management. It gives particular emphasis to the two technologies used in this thesis, namely the CORBA Interface Repository and the MOF, considering their application to the description of middleware platforms.

Chapters 4 and 5 present the design of the Meta-ORB reflective middleware platform, which represents the approach proposed in the thesis. The structure of the presentation reflects the overall organisation of the approach. In particular, Chapter 4 presents the meta-model of the platform, which defines the constructs used to build particular platform and application configurations. Importantly, the definition of the meta-model includes a prescription of the framework for managing and using meta-information defined according to its meta-types. This is followed by Chapter 5, which presents the meta-level design of the platform, in terms of its overall organisation and its integration with the meta-information management framework. Importantly, the meta-level is defined using the constructs of the meta-model defined in the previous chapter. This enables its design to be defined in an abstract way, without committing to a particular structure or functionality.

Chapter 6 then discusses the implementation of a proof-of-concept prototype, which realises the design presented in the previous two chapters. In particular, a concrete implementation of the meta-information management framework is described, in terms of a type repository, along with an infrastructure of services to support the meta-model (such as factories for the instantiation of the meta-model constructs). The chapter then presents one possible realisation of the meta-level design, by defining concrete meta-object protocols and their implementation in terms of meta-objects. The chapter also discusses the implementation aspects of the integration between reflection and meta-information management.

Chapter 7 follows with an evaluation of the proposed approach, aiming to demonstrate its practicality and major benefits. A simple application scenario will be used, in order to analyse the performance characteristics of the approach, as well as its more abstract properties, notably usability and the fulfilment of important requirements of emerging application areas.

Finally, Chapter 8 highlights the results and contributions of the thesis, also assessing the several aspects of the proposed approach from the point of view of the reflective middleware requirements introduced in Chapter 2. The chapter also considers important areas for potential future work.
Chapter 2  Fundamentals of Reflective Middleware

2.1 Introduction

In this chapter, the underlying principles of reflection and meta-level architectures are reviewed, with a focus on the reflective techniques that are useful in the context of middleware. Representative reflective architectures are described, showing applications of these principles in the areas of programming languages, operating systems and distributed systems. The chapter then discusses the use of reflection as an architectural principle for middleware, based on the requirements of emerging areas of application. The aim of this discussion is to identify design principles for reflective middleware and also to contrast these principles with currently established middleware technologies and standards. Examples of reflection-enabled middleware architectures from the literature are then examined and compared with the same design principles. Finally, the chapter considers alternative approaches to reflection, notably open implementation and aspect-oriented programming, in an attempt to show how they relate to reflective meta-level architectures.

2.2 Principles of reflection and meta-level architectures

2.2.1 Foundations

The fundamentals of reflective computing systems were introduced by B. C. Smith and can be summarised by his reflection hypothesis [Smith 1982], which argues that a system can be made to manipulate representations of itself in the same way as it manipulates representations of its application domain. Such a system is said to have a self-representation (also referred to as a meta-representation), which can encompass both its state and behaviour. In addition, if there is a relationship of causal connection [Smith 1984; Maes 1987] between the self-representation and the actual state and behaviour of the system, meaning that changes in one have corresponding effects in the other, the system is said to be reflective. The self-representation can thus be used for inspection and adaptation of the system's internals. It is important, though, to distinguish between systems with a reflective architecture and systems with reflective
facilities [Maes 1988]. Whereas in the former reflection is an inherent principle that permeates the design, the latter use ad hoc approaches, where the self-representation only considers a few aspects of the system, not necessarily in a uniform way.

The architecture of a reflective system is usually structured in levels, where the bottom level, known as base-level, deals with computation about the domain of application, whereas the levels above it, known as meta-levels, perform computation about the system itself. More precisely, each meta-level is concerned with the representation and manipulation of the level below it (which is its relative base-level), giving rise to the notion of a reflective tower of meta-levels, as illustrated in Figure 2.1. In principle, as with recursive procedures, this tower can have infinite levels. In practice, however, the reflective recursion is ended by implementing the topmost level as a hard-coded primitive interpreter, which is fixed and not subject to the reflection process. Furthermore, the use of techniques such as the lazy creation of meta-levels means that, in practice, only a few levels are actually present [Smith 1984]. Typically, the reflective tower is realised in a meta-circular way, with each level (except the topmost one) implemented in the same language as the level it reifies.

Figure 2.1 – Reification, reflection and the tower of meta-levels.

As illustrated in Figure 2.1, the act of a meta-level exposing the internals of its (relative) base-level is known as reification. This corresponds to the establishment of an explicit representation of the base-level system and its internal implementation, in terms of programming entities that can then be manipulated at runtime. Modifications to this self-representation result in corresponding changes to the reified elements of
the base-level, a process known as reflection or absorption. Importantly, this process of reification and reflection respects the causal-connection property described above.

In general, a reflective architecture that includes the above principles in one form or another is referred to as a meta-level architecture. Such architectures inherently exhibit a basic degree of separation of concerns [Hursch and Lopes 1995], which helps to simplify design and implementation, by considering application core concerns at the base-level, separately from the non-functional concerns, which are handled at the meta-level. As a consequence, not only the design and implementation of applications becomes simpler, but also the non-functional aspects realised at the metalevel can potentially be reused among different applications in a seamless way [Cointe 1996]. In addition, the meta-level in such architectures is typically structured in terms of well-defined entities and, given a particular base-level element, the set of meta-level entities comprising its self-representation is known as its meta-space.

The above principles of reflective systems are sufficiently generic to allow different interpretations, resulting in different sorts of reflective architecture. The next subsection presents an abstract classification of reflection, according to a few dimensions that are useful to understand existing reflective systems and architectures.

### 2.2.2 Styles of reflection

#### Behavioural versus structural reflection

The first of these dimensions relates to the distinction between structural reflection and behavioural reflection, initially conceived in the context of programming languages [Malenfant et al. 1996]. Structural reflection is defined as the ability of a language to provide a complete reification of the program currently executing, together with the abstract data types that are part of the program. On the other hand, behavioural reflection (also referred to as computational reflection [Maes 1987; Watanabe and Yonezawa 1988]) is the ability of a language to provide a complete representation of its own semantics, in terms of the internal aspects of its runtime environment. Hence, while structural reflection usually deals with the functional properties of a system, behavioural reflection is typically concerned with the non-functional properties. Importantly, these two styles of reflection are complementary to each other, with many reflective architectures providing both.
Chapter 2 – Fundamentals of Reflective Middleware

Procedural and declarative reflection

Reflective architectures can also be classified as either following a procedural or a declarative approach [Maes 1988]. In *procedural reflection*, the self-representation of a system is (or directly manipulates) the system's own implementation, meaning that whenever reflective computation alters the self-representation, the causal connection property is automatically guaranteed. In contrast, in *declarative reflection*, the self-representation is distinct from the implementation of the system, and instead consists of statements and properties about its behaviour and structure. This implies that causal connectivity has to be explicitly maintained. As a consequence of the declarative approach, only a limited number of aspects can be reified in practice, making the approach somewhat inflexible. On the other hand, procedural reflection allows virtually any element of a system to be reified and manipulated, also enabling completely new behaviour to be introduced into a system. Note, however, that a combined approach, providing a declarative interface on top of procedural reflection, is also possible.

Intensional and extensional reflection

Another dimension for the classification of reflection techniques relates to the notions of *intension* and *extension*. Reflection at the intensional level allows the base-level to be reified in terms of the types of its entities, therefore implying that all instances of the same type are handled at once, with no way of separating their representations at the meta-level. In contrast, reflection at the extensional level is concerned with the manipulation of individual base-level entities (i.e. the extent of types), through their own private meta-level representations.

A related issue refers to the distinction between the *meta-class* and the *specific meta-object* models [Ferber 1989]. In the former, all instances of a class share the same meta-space (which is the class itself and its meta-class), whereas in the latter, each base-level entity has its own meta-space. However, this is different from the distinction between intensional and extensional reflection. In particular, intensional reflection deals with a type as a predicate that states the properties of its instances (in the sense of RM-ODP [ITU-T/ISO 1996]), while the meta-class model handles actual implementations of a type (i.e. the same type can be realised by more than one class).
2.2.3 Structure of the meta-level

A well-defined meta-level structure is an important ingredient to facilitate the use of a reflective architecture. This is particularly an issue with procedural reflection, due to the multitude of aspects that can be handled. For instance, consider the procedural reflection capabilities natively available in Lisp dialects, which allow a program to manipulate its own code as data. Due to the lack of higher-level structures (code is treated in terms of abstract syntax trees), reflective computation has to funnel through the basic *eval* functions of Lisp [Foote 1990], making it difficult to recognise the specific aspects of the system that need to be manipulated. What is needed, thus, is a meta-level that allows each of the concepts of the system to be easily identified, in terms of discrete elements that can be handled separately from each other.

The *object-oriented* paradigm provides a clean way to structure the meta-level representation, in terms of principled mechanisms [Gabriel *et al.* 1991]. For instance, in the ABCL/R language [Watanabe and Yonezawa 1988], the several aspects of an object’s structure and behaviour are explicitly modelled by distinct objects of its meta-space. In fact, as observed in [Kiczales *et al.* 1991], there is an important synergy between reflection and object-orientation. Reflection makes it possible to open up a system’s implementation without revealing unnecessary details. Object-orientation, on the other hand, allows the resulting model of the system’s structure and behaviour to be locally and incrementally adjusted. In other words, it allows the distribution of the reflection mechanisms and interfaces among multiple meta-level entities [Foote 1990]. Note that similar benefits can also be achieved by using *components* as the underlying concept to structure the meta-level, as will be considered later in this thesis.

Regarding terminology, in *object-oriented reflection*, the entities that populate the meta-level are called *meta-objects* [Kiczales *et al.* 1991], while those entities at the base-level are known as *base-level objects*. Thus, while the interfaces of base-level objects provide an object protocol for access to application functionality, the interfaces of meta-objects provide a *meta-object protocol* (MOP), which allows reflective access to the implementation of the system [Kiczales *et al.* 1991]. Importantly, the same object model should be employed at both base- and meta-level, meaning that reflection can be re-applied at the meta-level itself.
Chapter 2 – Fundamentals of Reflective Middleware

2.3 Representative reflective architectures

2.3.1 Outline

The level of interest in reflection is confirmed by the number of applications of the concept in distinct areas, from programming languages to middleware. This section considers some representative examples of reflection in programming languages, operating systems and distributed systems. The use of reflection in middleware is examined in more detail later in the chapter, as it is a main concern of the thesis.

2.3.2 Reflection in programming languages

Reflection techniques have been widely used in programming languages as a means to enable the explicit manipulation of the structure and behaviour of programs. Early examples can be found in the Lisp and Smalltalk communities. The dynamic nature of such languages enables code to be explicitly handled as program data, constituting a basic form of structural reflection. Lisp-related languages usually explore the *meta-circular* definition of the interpreter, which is implemented in the same language as the user-level programs, thus enabling behavioural reflection by explicit manipulation of the implementation of the language. Examples are CLOS [Gabriel *et al.* 1991], 3-KRS [Maes 1987] and ObjVlisp [Cointe 1987]. In contrast, in languages belonging to the Smalltalk family, such as Smalltalk-80 [Goldberg and Robson 1983], such ability is usually realised by allowing some level of access to the virtual machine, by means of hooks that enable different aspects of the language semantics to be handled and adapted [Foote and Johnson 1989].

More recently, reflective techniques have been employed in languages such as Java and Python. In Java, native reflective facilities are available through the java.lang.reflect class library [Sun Microsystems 2000], which allows introspection (i.e., inspection) on the structure of attributes and methods, as well as dynamic access to these features. In addition, limited support for behavioural reflection has been added to Java, based on the notion of dynamic proxies. This enables the interception of methods calls in order to, for instance, add non-functional properties. Further, Java’s dynamic proxies can also be used to dynamically add interfaces to objects, thus characterising a level of structural adaptation. The Python language [Lutz 1996] also provides native reflection facilities, which allow full dynamic access to the meta-
representation of classes and objects, enabling runtime introspection as well as adaptation. For instance, it is possible to add or remove methods to classes or to individual instances, as well as to change the class of an instance.

In addition, a number of language extensions, especially for Java and C++, have been proposed, in order to provide more comprehensive reflection support. The level of support can be roughly classified according to the point in the lifecycle of a program where reflection is available to the programmer. At one end of the spectrum, OpenC++ [Chiba 1995] and OpenJava [Tatsubori et al. 2000] adopt a compile-time approach, which allows a level of control over the compilation process, thus enabling customisation of the language semantics. Meta-level constructs, however, are not available after the compilation process (although in OpenJava, they are available for runtime structural reflection only). At the opposite end, reflection has been implemented as a runtime feature in languages such as Iguana [Gowing and Cahill 1997] (a reflective architecture that has been implemented both for C++ and Java [Redmond and Cahill 2000]), MetaXa (formerly MetaJava) [Golm and Kleinoder 1997], Reflective Java [Wu and Schwiderski 1997] and Guaraná [Oliva et al. 1998].

Generically, meta-level constructs (meta-objects and/or meta-classes) are used for the reification of language concepts, and interception mechanisms provide hooks where new behaviour can be inserted or modified in terms of such meta-level constructs. In addition, reflective extensions to Java have also explored other points in the lifecycle of programs. For instance, Javassist [Chiba 2000] and Kava [Welch and Stroud 1999] are based on byte code transformations, normally effected by a modified class loader, in order to allow, respectively, load-time structural reflection and dynamic interception of key features of program behaviour.

2.3.3 Reflection in operating systems

The need for customisation and dynamic adaptation of their execution environment and services has motivated the development of operating systems that apply reflection as a basic design principle ([Kiczales et al. 1993; Kiczales 1994; Gowing and Cahill 1995]). In addition, according to [Yokote 1992], the separation between base- and meta-level is beneficial to object-oriented operating systems, as it overcomes the limitations imposed by encapsulation when a kernel object needs unrestricted access to the internals of another object. This is particularly true in the case of object
managers and debuggers; their implementation as meta-objects naturally allows access to all required information about the state of (base-level) objects in the system.

A representative example of an operating system employing reflection as a fundamental architectural principle is Aperios (formerly known as Apertos), from Sony [Yokote 1992; Lea et al. 1995]. In the Aperios framework, all aspects of the system are implemented at the meta-level, in terms of meta-objects. The behaviour of a user object is then reified (and indeed implemented) by its associated meta-space, which can be seen as the virtual machine for the object. The approach is thus characterised by procedural reflection. In addition, as meta-objects are themselves objects, they can also have meta-spaces defining their behaviour. The meta-space of a base-level object thus consists of a group of meta-objects organised in a hierarchy (or tower), which is ended by having its top level fixed and not subject to reification. Dynamic adaptation of services in this environment is then made possible by transparently migrating an object into a new meta-space that provides different system services and behaviour.

Another example of a reflective operating system is MetaOS [Horie et al. 1997], which is similar to Aperios, but adopts an explicit meta-interface to allow dynamic adaptation of meta-spaces, avoiding the need to migrate the base-level object to an entirely new meta-space. In addition, its meta-space is limited to two meta-levels. Yet another interesting example is CHEOPS [Schubert 1997], where structural information about the classes used at design time is retained at runtime, in the form of wrappers of their instances. Dynamic adaptation of the system is made possible by replacing such wrappers, in order to change the behaviour of the instances of a class.

### 2.3.4 Distributed reflective architectures

Several of the languages and systems mentioned above can be used to support distributed applications. OpenC++, for instance, enables an extension to C++ in which a class can be declared as distributed, meaning that runtime distribution support is provided for its instances. Another example is Aperios and its predecessor Muse [Yokote et al. 1989], which are qualified as distributed operating systems, as they support distributed resource management. Experiences such as these, as well as the observation that reflection is an essential mechanism to enable distributed systems
extensibility [Stroud 1993], have motivated significant research efforts towards reflective architectures that provide explicit representation of distribution aspects.

CodA/Tj [McAffer 1995; McAffer 1996] is a meta-level architecture (CodA) with a distributed object model (Tj). The architecture enforces separation of concerns via the usual distinction between base- and meta-level. Interestingly, CodA recognises the complexity of meta-levels for distributed systems, and addresses this issue by decomposing the meta-level into a number of different facets (or roles), one for each distinct reified aspect. Each facet is realised by a meta-component, which can be further reified according to the same principle. Typical facets include message sending and reception, mapping from messages into methods, execution behaviour, and object state. The CodA MOP then consists of the union of the protocols associated with each facet, which means that adding a new facet results in an extension to the MOP.

Another meta-level architecture that addresses the issues of reflection in distributed systems is AL-1/D [Okamura et al. 1992; Okamura 1995]. AL-1/D introduces the concept of multi-model reflection framework (MMRF) as a uniform way to decompose the structure of the meta-level. In the MMRF, the meta-level is structured in terms of a number of models, which represent distinct views of the system. Although in principle the number of models could be open, AL-1/D defines a fixed set of six models. The models deal with aspects such as the semantics of method calls, resources, object migration, low-level object representation and system statistics. In comparison to CodA’s facets, AL-1/D models are rather coarse-grained. Nevertheless, structuring a model in terms of finer-grained objects helps to reduce this disadvantage.

Importantly, the use of a MMRF, as in AL-1/D, (and, to a certain extent, facets in CodA) contributes to the uniformity of the meta-level, as all base-level objects have a meta-space structured along the same principles. In addition, besides coping with the complexity of implementing a meta-level for distributed systems, it also facilitates the understanding of the reflection capabilities, from the programmer’s perspective, as the overall MOP is partitioned into more manageable units.

2.3.5 Other areas of application

The systems briefly described above give a sample of the areas in which reflection has been applied. Other important areas of application include active networks [Villazon 2000], virtual network architectures [Campbell and Kounavis 2000], mobile
computing [Efstratiou and Cheverst 2000], mobile agents [Ledoux and Bouraqadi-Saadani 2000] and fault tolerance [Buzato et al. 1997; Killijian et al. 1998]. The use of a coherent core of reflection concepts in such distinct areas confirms the generality of reflection as an architectural principle for software systems. Reflection can thus be considered as a universal approach to open up and expose the internal implementation of systems, in a way that enables their inspection and adaptation. In particular, this thesis is concerned with this use of reflection in distributed systems middleware. The next section considers the rationale for this and discusses important features that should be present in reflective middleware architectures.

2.4 Reflective middleware

2.4.1 Overview and motivation

The overall rationale for reflective middleware comes from the need to open up the platform implementation, in order to allow the customisation and runtime adaptation required by dynamic applications (such as those discussed in the Chapter 1). Existing middleware technologies have recognised the need to address this problem, although tackling it with a different approach, by adding flexible features on top of their core architectures. Despite the usefulness of these features, as will be seen in section 2.5, the degree of support for customisation and dynamic adaptation is only partial, not covering all aspects of the design and the different phases of a platform's lifecycle. This is mostly due to the inherent “black-box” nature of these technologies, which limits the extent to which elements of the design can be opened and exposed to the programmer. Reflection, on the other hand, offers a truly generic solution to the above problem, as it enables a principled approach to the design of middleware in a way that naturally renders itself to openness [Blair and Papathomas 1997]. In addition, the use of reflection enables the different aspects of a platform to be manipulated and adapted in ways that were not anticipated during its design.

In general terms, reflective middleware refers to the use of a causally connected self-representation to support the inspection and adaptation of the middleware system [Coulson 2000]. Thus, the same reflection techniques used in traditional areas (such as programming languages) apply to middleware as well [Cazzola et al. 2000]. Nevertheless, the use of reflection techniques in the design of middleware introduces
new challenges, which are either not found or less important in such traditional areas. In the remainder of this section, these challenges are presented, in the form of a set of requirements and principles that are of particular importance in the design of reflective middleware architectures. These requirements and principles refine the meaning of reflective middleware from the point of view of this thesis. In addition, they constitute a framework for evaluating the reflective middleware architectures examined in section 2.6, as well as the particular approach proposed in this thesis.

It is important to note that most of the issues that appear in the design of conventional middleware platforms also hold for reflective middleware. In particular the following concerns usually need to be observed [Coulouris et al. 2001]: openness (in the face of heterogeneity), scalability, concurrency control, distribution transparency, security and quality of service. As such issues have been extensively considered in the literature, addressing them is not a main theme in this thesis, although their importance should be recognised in the proposed approach.

2.4.2 Evaluation framework

Overall design issues for reflective middleware

1. Modular platform infrastructure. This is essential in order to facilitate a clear identification, at runtime, of the services and components of a platform. It also enables each of these elements to be manipulated independently from the others.

2. Language and system independence. In order to enable portability of applications that use reflection functionality, it is important that a MOP for reflective middleware is defined at a language-neutral level. Thus, the use of reflection capabilities of specific languages or operating systems should be discouraged.

3. Approach to separation of concerns. This issue is related to the usual distinction between base- and meta-level. In the context of reflective middleware, special consideration is required to clearly define the role of each level, with respect to both platform infrastructure and applications. This is further discussed below.

4. Access to the meta-level. Two basic forms of access to reflection mechanisms are possible [Maes 1988]: implicit access (when the flow of control is transparently transferred to the meta-level) and explicit access (where the program explicitly decides when to invoke the meta-level). This issue is further considered below.
5. **Granularity.** This issue is concerned with the units of abstraction that can be directly subject to reflection and, especially, with the flexibility for the programmer to choose the most appropriate abstraction level for each particular case. For instance, reflection can be applied to the whole platform as a unit or, alternatively, to individual interfaces of primitive platform components.

6. **Scope of reflection.** Reflective functionality can be applied either to individual objects or to arbitrary collections of objects in a middleware platform. The former tends to be more appropriate in the context of dynamically adaptable middleware, as individual object instances are likely to be independent from each other, at least for the purposes of adaptation. Nevertheless, there are also cases where reflective operations should consider entire collections of objects. An example is the use of reflection to manage object group services, such as in [Saikoski et al. 2000]. Another example is service type modification, where all instances of the same service type need to be updated at once.

7. **Pervasiveness of reflection.** This issue is orthogonal to the previous two and refers to the range of aspects of a middleware platform (e.g. marshalling, synchronisation, communications protocols, and the several distributed systems services) that can be reified with the reflection mechanisms. Ideally, all aspects of interest in a given context should be amenable to manipulation via reflection.

8. **Uniform reflection model.** This issue crosscuts the previous three and means that a consistent set of reflection mechanisms and interfaces (cf. MOP) should be applicable irrespective of granularity level, scope or aspect of a middleware platform. It is also important that the same semantics is retained in all cases.

9. **Unified approach to configurability / re-configurability.** Essentially, facilities for static configuration (which enable the initial selection of the middleware components required in a particular scenario) should be highly integrated with the mechanisms used for dynamic reconfiguration. In particular, both should be based on a unified terminology and use a consistent set of meta-level constructs.

10. **Safety of reflective computation.** The ability to dynamically change the implementation of a middleware platform introduces the need to manage the correctness and coherency of the platform, during and after the reflective changes. In addition, issues of security, such as access control and authentication, can be
considered, in order to prevent unauthorised access to the reflection mechanisms that affect critical aspects of the platform.

11. Management of meta-level complexity. Meta-levels for middleware platforms tend to be highly complex, due to the large number of aspects that need to be represented and the interactions among them. In addition, the need to handle such aspects in a distributed environment only adds to the complexity. Alternatives to handle this problem are considered below.

12. Management of meta-information. Meta-information permeates the lifecycle of reflective middleware platforms, from analysis and design to runtime. In addition, the nature and contents of the meta-information that emerges in each phase is essentially the same, making it useful to have a common framework that allows such meta-information to be managed in a unified way. This helps bridging the gap between the different phases of a platform's lifecycle, enabling design decisions to be exposed at runtime, so that they can be inspected and evolved using the MOP. In addition, it contributes to fulfil the requirement presented in item 9 above. Meta-information management will be considered in Chapter 3.

13. Performance. This is a familiar issue in reflective architectures, and its main concern is the extra overhead imposed by the reflection mechanisms, which should not significantly impact the performance of applications as perceived by users. Common solutions that apply in the context of middleware include lazy instantiation of meta-objects, and the use of the meta-level itself and its facilities in order to improve the overall system performance (such as in [Itoh et al. 1995]).

While most of these issues follow quite naturally, some require further discussion to clarify their meaning and the options available, as seen in the next paragraphs.

Approach to separation of concerns

Separation of concerns is an essential issue in reflective systems. In the traditional reflection literature, it takes the form of the distinction between base- and meta-level, to imply that user programs belong in the base-level, while their underlying implementation (e.g., the interpreter) belong in the meta-level [des Rivieres and Smith 1984]. While this is the natural approach in programming languages, in middleware a number of alternatives to separation of concerns are feasible. The usual approach (see
section 2.6), which directly derives from the traditional interpretation, is to regard distributed applications as the base-level and the supporting middleware platforms as their meta-level. In this way, besides providing for the reification of the platform internals, meta-level entities (e.g. meta-objects) also constitute their actual implementation, meaning that the reflection mechanisms are instantiated by default.

An alternative approach is to consider the meta-level as an orthogonal plane to the middleware and application layers, as shown in Figure 2.2.

![Figure 2.2 – Approach to separation of concerns: Middleware as part of the base-level](image)

In this approach, both application and middleware have their usual functionality defined at the base-level, whereas the entities at the meta-level provide the facilities to reify, inspect and control such functionality. Viewing reflective middleware in this way enables a cleaner and unified model for reflective programming (see issue 8 above), as the same reflection capabilities apply to both middleware and applications. This approach also has the advantage that reflection mechanisms can be instantiated on demand (as the platform implementation does not inherently depend on the meta-objects that reify it), thus avoiding unnecessary overheads if they are not required.

Access to the meta-level

*Implicit access* to the meta-level enables individual instances of computation to be automatically reified, in order to have their execution controlled at the meta-level. This mode of access is essential for behavioural reflection. The usual way to implement it is by intercepting the interactions between base-level entities, transferring the control to the meta-level, which may introduce additional properties
and, afterwards, return the control to the base-level for the normal computation to resume.

In addition, it is also important to provide *explicit access* to the meta-level, in order to enable the user (meta-programmer) to configure the structure and behaviour of the system (for instance, to modify the structure of base-level objects or to change the way meta-objects react to implicit access). This kind of access is normally realised via an explicit meta-object protocol, distributed across a number of meta-interfaces.

In addition, a hybrid form of access can be achieved by having an automatic mechanism which, based on events and adaptation policies, determines the need for adaptation and makes calls to the meta-level in a transparent way.

**Management of meta-level complexity**

Techniques to improve separation of concerns are essential in order to tackle the multitude of aspects that compose a middleware platform. A common approach is to structure the meta-level in a modular fashion, with the use of objects (or components) to represent the individual reified aspects. However, although the modular decomposition of the meta-level helps to simplify the design of reflective middleware, the complexity and number of aspects that need to be treated can easily lead to confused meta-space configurations. It is therefore important to adopt a well-defined meta-level decomposition strategy, encouraging a uniform programming model across different instances of a platform. In addition, as proposed in [Okamura 1995], programmers naturally tend to use distinct views when dealing with the meta-level. These views can be identified and used as the foundation for such a uniform meta-level decomposition strategy, enabling the programmer to focus on the particular meta-level aspects of interest, in a way that is naturally isolated from other aspects.

**2.5 Reflection facilities in current middleware technologies**

**2.5.1 Overview**

This section examines representative middleware technologies, namely CORBA, COM/DCOM, and .NET, in order to identify the reflection-like features that have been included in their recent releases. It also examines the ISO RM-ODP framework, comparing its notion of openness with the notion of open implementation in reflective
architectures. The main aim is to show that the need for reflection is slowly being acknowledged by the middleware standard makers, as well as to demonstrate the problems associated with the lack of a more comprehensive reflection approach (in comparison with the issues discussed in section 2.4). Note that another technology that could be examined is Java Remote Method Invocation (RMI) [Sun 2000]. However, as the reflection features supported in Java RMI are essentially those defined for the standard Java environment (discussed in 2.3.2), it will not be further considered.

2.5.2 OMG CORBA

CORBA (Common Object Request Broker Architecture) [OMG 2001a] is the distributed object middleware standard from the Object Management Group (OMG) consortium. It defines an architecture for the mediation of requests between objects in a distributed system. The architecture is structured around an object request broker (ORB), which provides the basic interaction mechanisms, and a set of interfaces to access its services (either generic interfaces, such as the ORB interface and the dynamic invocation interface, or application-specific interfaces, such as stubs and skeletons). CORBA enables inter-language interoperability by providing a standard interface definition language (IDL) for the description of the interfaces of objects. Standard mappings from CORBA IDL to different programming languages enable clients and servers (objects) to be independently implemented in the language of choice for each one. On the other hand, operating system and hardware heterogeneity is resolved by making applications depend only on the interfaces provided by the ORB, thus achieving interoperability and a certain level of portability.

Initially, the philosophy adopted by the OMG was strictly based on a “black-box” approach, where open standards only applied to the interfaces of the ORB, while the internal details were left for each vendor to decide. This philosophy resulted in proprietary ORB implementations, making it impossible to have a consistent approach for platform customisation (although some vendors have long provided mechanisms to configure the ORB, such features are vendor-specific, leading to non-portable applications). While this black-box approach still largely applies to the most recent releases of the standard, from CORBA 2.3 a few aspects of the ORB have been opened for customisation. The discussion below, based on CORBA 3.0, considers the most notable reflection-like features that have been adopted as part of the standard.
Chapter 2 – Fundamentals of Reflective Middleware

The Portable Interceptors specification [OMG 2001b] enables the customisation of CORBA implementations by allowing extra services to be added to the core behaviour of the ORB. More specifically, interceptors implementing different services can be added at the level of requests (in order to intercept each invocation or reply to an interface and execute pre- and post-processing) or at the point of object creation, allowing the customisation of the creation policies and the components of IORs (interoperable object references). However, the use of interceptors is limited, as it is neither possible to add interceptors at arbitrary points in the ORB implementation, nor it is possible to define interceptors at different scopes other than the entire ORB instance. In addition, there is no native support for the structured composition or for the dynamic installation of interceptors. Arguably, the interceptor programmer can implement further interception functionality to overcome some of these limitations. However, the lack of standardisation limits the use of any such solution.

The other notable customisation feature of CORBA is related to the use of policy objects, which allow a level of control over the behaviour of some internal services of the ORB. Policies can be used to customise aspects such as the POA (Portable Object Adapter), asynchronous messaging, security, transactions and real-time. However, once a policy has been installed (e.g. at the time an object reference is created), it cannot be overridden, thus not enabling dynamic adaptation.

Such customisation abilities of CORBA have been considered as a form of reflection by some researchers (e.g. [Wang et al. 2001] and [Wegdam et al. 2000]). However, as they do not encompass all of the aspects of the ORB and do not enable arbitrary and dynamic adaptations, they are not sufficient to qualify CORBA as a reflective middleware architecture. From the perspective of this thesis, they are instead regarded as ad hoc (and static) reflection facilities.

2.5.3 Microsoft COM, COM+ and .NET

COM is the component object model from Microsoft [Microsoft 2000a]. It is based on the need to evolve application and system configurations by changing individual components independently from each other. Notably, new interfaces can be added to components (e.g., as versions of existing interfaces) and entire component implementations can be replaced without affecting existing clients, thus resembling structural reflection. These goals are mainly achieved with the principles of binary
encapsulation (enabling the machine code representation of components to evolve independently from their clients) and binary compatibility (meaning that machine code can be reused across development environments). Such features are enabled by separating the implementation of components from the definition of their interfaces, with the use of an interface definition language, in a similar way as in CORBA. However, the evolution of component configurations can only be achieved in a static way, and a meta-representation of such configurations does not exist at runtime. In addition, the fact that COM relies on binary code means that interoperability and portability cannot be fully achieved beyond the Intel/Windows environment.

The COM extension for distribution, known as DCOM [Microsoft 2000c], enables the introduction of customisations in its remoting mechanisms. This is achieved through the notion of custom marshalling supported by COM, which, similarly to interceptors, allows the adaptation of the proxy mechanism associated with the interfaces of an object. In this way, features such as the interaction protocol can be customised, although again in a static way.

COM+ [Microsoft 2000b] is the intended successor for COM, adding enhanced support to distribution, such as transactions. COM+ further proposes the notion of interception, which, similarly to CORBA, enables external services to extend the native COM+ services. The installation of COM+ interceptors can be made either by means of attributes, which are used to qualify classes, interfaces and methods (with properties that are realised through interceptors), or by using a programmatic interface [Box 1999]. In principle, it is feasible to use this interface to perform on-the-fly changes on the behaviour of an object, interface or method. However, similar to their CORBA counterparts, COM+ interceptors are limited to specific platform aspects.

Finally, the .NET framework [Microsoft 2000e] is the latest member of the Microsoft family of middleware technologies. Its aim is to provide a universal programming model for web services and applications. Notably, the framework is strongly based on the use of meta-information and support is provided for structural introspection, enabling the runtime discovery of the types of components. In addition, .NET takes the idea further and uses meta-information as a cornerstone to enable language-level interoperability. This is done by defining a language-independent type system, which supports all the runtime features of the platform (such as dynamic linking and remote method invocation) and enables cross-language development.
Interestingly, every .NET component or component assembly incorporates, as part of its runtime image, extensive meta-information that describes it according to this type system. Other uses of this meta-information are to enable remote method invocations (with the use of introspection for the automatic generation of proxies) and to resolve the (compile-time) dependencies of an assembly of components in relation to other assemblies, allowing the latter ones to be loaded on demand. The main limitations of .NET, again, are related to its dependency on Microsoft environments, as the common type system approach only works in the Windows family of operating systems. In addition, .NET lacks clear support for dynamic adaptation, as the meta-information maintained at runtime, although suitable as a self-representation, is not causally connected with the respective components.

2.5.4 RM-ODP

Differently from the technologies examined above, the RM-ODP (Reference Model for Open Distributed Processing) [ITU-T/ISO 1995d] standard provides an abstract framework for the development of distributed systems in an open services environment. This framework does not imply any particular implementation, but rather offers a generic model that can be interpreted and specialised by concrete middleware standards, such as, for instance, the OMG-based standards. RM-ODP is based around the concept of multiple viewpoints, which enables the development of ODP systems from different perspectives. Five viewpoints comprise the framework, along with their respective viewpoint languages:

- **enterprise viewpoint**: focuses on the business rules and policies governing a system, along with the roles played by the system, its users and the environment;
- **information viewpoint**: concerns the representation and semantics of the entities of information that need to be stored and processed in the system;
- **computational viewpoint**: describes an ODP system as a composition of objects, which interact through well-defined interfaces, in a distribution transparent way;
- **engineering viewpoint**: focuses on the mechanisms and functions that are required to support the distributed interactions between objects in an ODP system;
- **technology viewpoint**: deals with the particular technologies to realise an ODP system and its components.
In addition, RM-ODP prescribes a set of functions required to support open distributed processing, in terms of management, coordination of distributed computation, information storage and maintenance (repository), and security. It also provides a set of distribution transparencies, designed to hide the details of distributed computation and interaction (such as access, failure, location, replication and transaction) from the developers and users of ODP systems.

Interestingly, according to the standard, flexibility regarding evolution and dynamic reconfiguration is one of the crucial properties of an ODP system [ITU-T/ISO 1995a]. The basic prerequisites to support this property are provided by the engineering viewpoint language, which enables individual instances of a platform to be configured at the level of individual components, functions and transparencies, thus supporting a degree of configurability. In addition, the information viewpoint enables meta-information about the configuration of an ODP system to be described (such as in the case of the binding framework [ITU-T/ISO 1998b]) and potentially maintained at runtime (with the help of the ODP repository functions). However, no explicit support is defined for the use of such meta-information in a way that is causally connected with the engineering of a platform, hindering the provision of dynamic adaptation.

Therefore, although RM-ODP can serve as a basic framework for the definition of a reflective middleware architecture, the standard itself cannot be considered reflective. This lack of reflection in the model is also noticed in the definition of openness in RM-ODP, which is based on the properties of portability and interoperability of distributed system components in a heterogeneous environment. This is in contrast with reflective middleware, where this notion also encompasses implementation openness.

### 2.5.5 Discussion

It can be noticed from the above considerations that, as mainstream middleware technologies evolve, there is a clear trend towards acknowledging the need for flexibility. Interestingly, all of the platforms and standards considered above are beginning to provide some form of support for reflection-like capabilities. However, such capabilities are not typically an intrinsic part of the architecture, thus limiting their applicability. In addition, despite the flexibility that such capabilities enable, the issue of dynamic adaptation, especially considering arbitrary aspects of middleware
platforms, still remains open in such technologies. The provision of adequate treatment to this issue requires a more radical approach to the architecture of middleware. In particular, the prevailing understanding of openness in traditional middleware architectures (as defined in RM-ODP), lacks the important concept of implementation openness. As seen before, this concept is crucial to achieve the dynamism that emerging applications require. Section 2.6 presents some research initiatives that are based on this premise, while Chapters 4 and 5 present the particular approach proposed in the thesis.

Finally, a mention must be made here to the introspection facilities available in these architectures, notably the CORBA Interface Repository, COM Type Libraries and the RM-ODP Type Repository Function. These facilities have been considered as a form of reflection in the literature [Emmerich 2000], as they enable the dynamic discovery of meta-information describing the entities in a platform environment. In the context of this thesis, however, such features do not qualify the platform as reflective, as there is no causal connection between the items of meta-information and the corresponding entities in the platform. Instead, this kind of feature is considered as a form of (introspective) intensional reflection (see 2.2.2), in the realm of meta-information management, which will be examined in depth in Chapter 3.

2.6 Survey of reflective middleware research

2.6.1 Overview

This section examines representative related work on the use of reflection as a design principle for distributed systems middleware. The common theme is the use of reflection at a more fundamental level, in order to eliminate some of the crucial limitations of more conventional technologies, such as those highlighted in the previous section. The systems examined here were chosen to illustrate the main reflection techniques that have been used in the context of middleware, in particular: interception, meta-classes and component-based reflection. In addition, other related techniques, such as adaptive methods, dimensions of separation of concerns and communications reification are also illustrated. The section ends with a discussion where each of the examined architectures are contrasted with the reflective middleware design issues and principles introduced in section 2.4.
2.6.2 FlexiNet

FlexiNet is a configurable ORB architecture developed by APM, aiming at mobile object environments [Hayton 1997]. An important component of the architecture is an open binding framework, which defines the connection between clients and servers in terms of protocol stacks. This binding framework enables the configuration of the communications path between client and server, by means of an extensive use of some reflection techniques. Firstly, as FlexiNet is defined for a Java environment, it takes advantage of the Java Core Reflection API [Sun Microsystems 2000] to facilitate the generation of generic stubs, by means of direct access to the types of requests and dynamic method invocation. More important, however, is the use of reflection as defined by APM's Reflective Java extension [Wu and Schwiderski 1997], in order to allow for dynamically configurable method interceptors. Using this mechanism, client-side stubs and the server-side upper layer are instrumented with hooks that allow the plugging of interceptor meta-objects to implement customised non-functional properties (e.g., transactional properties, access control and replication).

The FlexiNet approach demonstrates the power of method interception for customising the non-functional aspects of middleware platforms, in a way that resembles behavioural reflection. It is more flexible than CORBA request interceptors in that it allows dynamic configuration and the use of interceptors at a more delimited scope. However, similarly to CORBA, the points at which interceptors can be inserted are restricted to the upper layers of the protocol stack, thus not allowing lower level mechanisms of the ORB (e.g. message-level serialisation and synchronisation) to be adapted. The only form of reflection that is available at the lower layers is structural introspection (using the Java Core Reflection API) for coupling protocol layers. In addition, considering the recent developments in Java Core Reflection, especially the new dynamic proxy feature, it can be argued that the FlexiNet framework could now be entirely implemented in standard Java. Although this does not invalidate the design of FlexiNet, it would make obsolete the use of APM's Reflective Java extension.

Another interesting aspect of FlexiNet corresponds to the FlexiBind framework [Hanssen 1997], which extends the original binding model with the idea of policies to control how bindings are configured. In addition, meta-policies are used to govern the process of binding activation and passivation, as well as the runtime selection of policies, enabling dynamic adaptation of entire binding configurations.
2.6.3 OpenCORBA

OpenCORBA [Ledoux 1999] is a reflective implementation of CORBA in NeoClassstalk [Rivard 1996], which in turn is an extension of Smalltalk with a meta-object protocol that allows the dynamic replacement of the class of an object and, importantly, the meta-class of a class. The reflective features of OpenCORBA are thus based on the idea of modifying the behaviour of a CORBA service by replacing the meta-class of the class defining that service. Two aspects of CORBA are reified and subject to this mechanism. First, it allows the dynamic adaptation of the behaviour of remote invocations by applying the above idea to the classes of proxies (stubs) and server templates (skeletons). By default, these classes are generated (according to the standard IDL to Smalltalk mapping) as instances of the meta-classes ProxyRemote and TypeChecking, respectively, which implement the standard CORBA behaviour. Replacing these meta-classes with custom ones therefore allows remote invocations with different non-functional properties. For instance, it is possible to implement a replication strategy at client side, or to introduce an optimised form of type checking of server invocations. The other aspect that can be subject to meta-class change is the creation of meta-information elements in the Interface Repository, allowing the adaptation of the strategy for validating the integrity of such elements.

The OpenCORBA approach thus allows for arbitrary customisations based on behavioural reflection. The facilities provided are an equivalent of request interceptors in CORBA, regarding the customisation of request behaviour. OpenCORBA, however, provides for dynamic customisation, whereas CORBA interceptors are static. In addition, such customisations can be made on a per-interface type basis, whereas CORBA interceptors apply evenly to all requests on an ORB instance. In general, however, the OpenCORBA approach is limited to the aspects described above (although the overall approach could apply more generically), thus not allowing, for example, the internal ORB mechanisms to be customised or adapted. Another limitation derives from the use of the meta-class approach, meaning that the scope of adaptation is a class, thus affecting all of its instances. For instance, as stub and skeleton classes are generated for each particular interface type, all the running instances of an interface type are affected by a meta-class change (as all of them share the same proxy and skeleton classes). As discussed in section 2.4, this scope may not be the most appropriate for middleware.
2.6.4 2K, dynamicTAO and UIC™

2K can be described as an adaptable, distributed operating system with an associated component framework [Kon et al. 1998]. The fact that it provides a comprehensive set of distribution services in a heterogeneous environment enables it to be considered as a middleware platform. Dynamic adaptability in 2K is based on reflection and on the concept of architectural awareness, making explicit the architectural structure of a system in a causally connected way. In addition, this configuration framework can be used to build customised instances of the platform, comprising only the services and components that are effectively needed.

The configuration framework of 2K is structured around a few meta-level concepts. Middleware configurations are defined in terms of prerequisite specifications, which consist of meta-information representing the components of the platform and the dependencies among them. These specifications are used by an automatic configuration service, which instantiates the required components and recursively evaluates their prerequisites in order to instantiate further components on which they depend. As part of this process, the configuration service may also instantiate, on a per-component basis, a component configurator, which is a meta-level object in charge of managing the runtime (dynamic) dependencies of the component in relation to other components, in order to determine the need for adaptation. Each component configurator is then responsible for keeping track of these dependencies as they evolve at runtime (by the addition or removal of dependencies). Interestingly, although component configurators provide a standard interface, their implementations can embed application-specific knowledge, for instance, about suitable adaptation strategies. In addition, the framework also prescribes a resource management service, which enables resource allocation (at instantiation time) and runtime monitoring, which can generate events to notify individual component configurators of resource changes, so that they can adapt their components accordingly.

The overall architecture of 2K follows a layered approach [Kon 2000]. The upper layer is populated by applications, running on top of a middleware layer, which provides uniform distribution services, supported by underlying reflective ORBs. These underlying ORBs provide the support for dynamic reconfiguration. The bottom (kernel) layer is then composed by traditional network operating systems and the underlying hardware. In principle, the framework described above can be applied to
each of these layers. In particular, the middleware layer consists of one or more reflective or customisable ORBs, which refine and implement this framework. Currently, this layer is realised by the dynamicTAO reflective ORB [Roman et al. 1999; Kon et al. 2000], although another option is the Universal Interoperable Core (formerly LegORB) component-based middleware platform [Roman et al. 2000; Roman et al. 2001].

DynamicTAO is a reflective extension of TAO [Schmidt et al. 1998], based on the ability to dynamically reconfigure the internal strategies (cf. the strategy design pattern [Gamma et al. 1995]) of the ORB, by plugging and unplugging strategy implementations on existing components. The structure of the platform is based on the 2K framework. It defines specialised component configurator classes, in order to provide for the dynamic management of different kinds of entities, such as particular ORB instances, ORB domains, and individual application servants. Each of these configurators defines a number of hooks for the installation of strategies, depending on the kind of component it configures. The interfaces of these configurators constitute the dynamicTAO MOP, with facilities for loading and installing new strategies, and for inspecting the state and structure of the reified components.

While the approach to customisation and dynamic adaptation of ORB components in dynamicTAO is based on strategies, UIC realises the same goals with a more explicit component-based approach, where the configurable entities are actual components. In addition, its emphasis is on environments with very limited resources, such as handheld computers. The architecture is based on a configurable ORB skeleton, which defines abstract components that represent customisation slots for the several ORB services. Besides allowing the flexible selection of concrete components for each abstract component, UIC also allows the customisation of the skeleton itself, by the addition or removal of abstract components, thus allowing a different set of services to be provided. Such customisations can be made either statically (at compile-time) and in an immutable way, or dynamically, by using a meta-interface similar to the dynamicTAO MOP. As a result, the basic ORB core can be specialised to provide different personalities (e.g., CORBA and Java RMI). In addition, a single UIC instance can have multiple personalities at the same time, enabling the integration of applications across a heterogeneous middleware environment.
The overall approach of the 2K framework represents a comprehensive treatment to the issues involved in dynamic reconfiguration of middleware. Problems such as the consistency of the ORB after reconfigurations are addressed, in order to avoid conflicts between the new and old configurations. Furthermore, a uniform reflection model is adopted, which can feasibly be applied to both middleware and applications. However, the reflection model is mainly targeted at the structural aspects of middleware. The lack of an explicit treatment of behavioural aspects means that they need to be dealt with in terms of structural reflection, potentially increasing the complexity of meta-programming. In addition, although the basic framework can be applied at different levels of granularity, the focus of the research is on reifying rather coarse-grained middleware components. This is mainly due to performance reasons, as it would be considerably inefficient to apply the framework to control components at a more detailed level.

2.6.5 DART

The Distributed Adaptive Run Time (DART) [Raverdy and Lea 1998] is a platform that enables applications to be aware of their environment, in order to handle runtime changes to both the application and its underlying system. It is based on a clear distinction between structural and behavioural reflection, through the use of adaptive and reflective methods. Adaptive methods are concerned with the adaptation of the functional aspects of an application (structural reflection), when the code of the application itself has to adapt to environmental changes. An adaptive method is basically a method with several distinct implementations, together with a selector, which chooses the right implementation to serve each call, depending on runtime configurable policies. On the other hand, reflective methods are methods whose non-functional properties can be adapted at the meta-level (behavioural reflection). The meta-space of a reflective method consists of a number of meta-level objects, implementing distinct non-functional properties, along with a reflector, which is responsible for the interception of method calls and for the invocation of the meta-level objects. Behavioural adaptation (e.g. due to mobility) is then achieved by changing the constitution of the meta-space. Note, however, that DART’s support for adaptation is based on the reflective facilities of the Open C++ v2 [Chiba 1995], which goes against the principle of language independence, discussed in 2.4.2.
2.6.6 RORB

RORB is a reflective ORB architecture that is part of the RODS distributed system [Chen et al. 1996]. It is a three-level reflective middleware architecture, where the meta-level controls and monitors the base-level, while the meta-meta-level provides for the monitoring of the meta-level. The reflection mechanisms reify a fixed set of aspects related to the application behaviour (proxying and server management) and to the platform internal structure (location control, marshalling and unmarshalling, connection control, session control and transport control). Each of these aspects is represented by a distinct meta-object, which can be dynamically changed (using its meta-interface) in order to modify the application behaviour. Extensibility of the meta-level is achieved by deriving new meta-object classes from basic ones that are pre-defined for each of the reifiable aspects. However, the set of reifiable aspects cannot be extended. Another interesting feature of RODS is the adoption of a two-dimensional reflective architecture (derived from the AL-1 language [Ishikawa 1991]), where one dimension models the reflection tower, while the other models the levels of abstraction, called floors: application, middleware and OS kernel. Thus, it is possible to use meta-objects (and meta-meta-objects) to reify application and middleware objects according to the same principles.

2.6.7 Quarterware

Early work on reflective ORBs at the University of Illinois has proposed the use of a limited form of reflection in order to allow the customisation of timeliness and fault tolerance aspects of a real-time ORB [Singhai et al. 1997]. Their approach is based on a component framework for middleware, called Quarterware [Singhai et al. 1998], where the different internal mechanisms of the ORB are realised in terms of components, as a way to facilitate customisation. A reflective interface is then provided that allows the programmer to install customised versions of these components. The following ORB mechanisms can be customised: method dispatch, memory and concurrency management, object creation and destruction, object reference management, and marshalling. Significantly, this work demonstrates the importance of a modular (component-based) structure of middleware for the provision of reflective interfaces that manipulate this structure. The main drawback of this approach, besides the limited range of adaptable ORB services, is the fact that it
requires one specific reflective interface for each adaptable aspect, potentially contradicting the principle of uniformity. Thus, as the number of adaptable aspects increases, new reflective interfaces must be added, making the programming model difficult both to manage and to understand.

2.6.8 mChaRM

The approach advocated in the mChaRM reflective middleware [Cazzola and Ancona 2000] proposes the use of channel reification [Ancona et al. 1995] to enable explicit control over multi-party communications. It is centred on a meta-level abstraction, the channel, which permits the handling of messages between objects in a session-based fashion, making possible to intercept method calls and introspect and adapt both their structure and behaviour. The main advantages of the approach are a global view of interactions among objects (in contrast to reifying each communications endpoint separately), and the fine level of granularity for reflective computation (based on each message and its structure). The main limitation is related to performance, due to the inherent overhead of message interception and transfer of control to the channel objects. In addition, structural reflection other than on channels (e.g. on objects that provide other middleware services) is not supported.

2.6.9 Discussion and comparative analysis

The tables presented in Appendix A provide a comparative analysis of the reflective middleware technologies examined in this section, contrasting their features with the evaluation framework introduced in section 2.4. Note that this is not an attempt to provide a classification of reflective middleware platforms. Instead, it aims at showing how each of the examined approaches answers the design requirements that characterise reflective middleware from the point of view of this thesis. The main conclusions from this analysis are summarised in Table 2.1 below.

As the analysis shows, most of the design issues are considered, in one form or another, by all the examined architectures. However, the degree to which each one is treated is usually less than complete, not exploring important aspects of middleware design and, thus limiting the benefits of reflection. This points towards the need for a more principled and architectural approach to reflection in middleware, which is the goal of this thesis. In addition, existing approaches to reflective middleware do not
consider the explicit and unified management of the meta-information handled by the reflective mechanisms. This is regarded as an important issue in this thesis and is one of the central concepts in the proposed approach.

Table 2.1 – Summarising the features and limitations of existing reflective middleware.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular platform infrastructure</td>
<td>although nearly all of the studied architectures exhibit a degree of modularity, it is usually not designed specifically to support the reflection mechanisms.</td>
</tr>
<tr>
<td>Language and system independence</td>
<td>the benefit of providing reflection as a middleware-level feature is generally acknowledged, although a few approaches use language-specific mechanisms.</td>
</tr>
<tr>
<td>Approach to separation of concerns</td>
<td>with the exception of 2K, all of the examined systems consider the middleware as the meta-level.</td>
</tr>
<tr>
<td>Access to the meta-level</td>
<td>the need for both implicit and explicit access is generally acknowledged.</td>
</tr>
<tr>
<td>Granularity</td>
<td>nearly all of them employ a rigid granularity level (again with the possible exception, in principle, of the 2K framework).</td>
</tr>
<tr>
<td>Scope of reflection</td>
<td>different scoping rules are exemplified in the examined architectures, but with the predominance of instance-based scopes (either per object or per channel), thus reinforcing the argument in favour of this choice for reflective middleware.</td>
</tr>
<tr>
<td>Pervasiveness of reflection</td>
<td>treatment of this issue is usually limited to a fixed set of aspects that can be reified, although in the case of 2K, the reflection framework can potentially be applied to any component or aspect of the platform (as a consequence of their adopted approach to separation of concerns).</td>
</tr>
<tr>
<td>Uniformity of the reflection model</td>
<td>this seems to be a general problem, as most of the examined systems employ distinct MOPs for distinct aspects of a platform.</td>
</tr>
<tr>
<td>Configurability and re-configurability</td>
<td>the need for dynamic reconfiguration is confirmed by all of the examined systems; support for static configuration is usually provided, although not always integrated with the dynamic reconfiguration facilities.</td>
</tr>
<tr>
<td>Safety of reflection</td>
<td>this is often seen as an important issue, although most systems lack a more comprehensive support (2K being the notable exception; see Appendix A); the usual approach seems to rely on the scope of the reflection operations to naturally limit any undesired side-effects.</td>
</tr>
<tr>
<td>Management of meta-level complexity</td>
<td>nearly all of the systems recognise this issue, although its treatment usually takes an ad-hoc form.</td>
</tr>
<tr>
<td>Explicit management of meta-information</td>
<td>this is a largely overlooked issue; although all the examined systems deal with meta-information in an implicit way, none of them provide a common framework for its management.</td>
</tr>
<tr>
<td>Performance-enhancing mechanisms</td>
<td>in the cases where this issue is considered, the usual approach is to avoid the overhead of reflection when it is not required; in the case of 2K, reflection is also used as a means for optimisation.</td>
</tr>
</tbody>
</table>
2.7 Alternative and complementary techniques

The open implementation approach [Kiczales 1994; Kiczales 1996] constitutes an alternative way to provide and expose the self-representation of a system. An object designed according to this approach would have at least two distinct interfaces: one to access its normal services (the base interface), and another to access its internal implementation (the meta-interface). According to [Kiczales and Paepcke 1996], open implementations can be used as a means for the design of meta-object protocols, by using reflection as the principle to access the internal implementation. Note, however, that, as pointed out by [DeVolder and Steyaert 1995], an open implementation can only be considered as having such reflective potential if its self-representation is realised in a meta-circular way.

Regarding separation of concerns, aspect-oriented programming (AOP) [Kiczales et al. 1997] proposes an alternative approach, based on identifying the several cross-cutting aspects involved in the construction of a system, and programming each one separately, possibly using a different language. Subsequently, a process called aspect-weaving enables the automatic combination of the aspect programs into a complete system. In this way, AOP considerably simplifies large systems programming. However, because the typical weaving process is static, aspect descriptions are not preserved at runtime, making it impossible to dynamically reify aspects. Nevertheless, as pointed out by [Malenfant and Cointe 1996], AOP and reflection can be seen as complementary techniques. In particular, AOP offers a useful abstraction principle to structure the meta-level, especially considering complex systems, such as middleware. An example is the multi-model reflection framework proposed in [Okamura 1995], where each model can be seen as a dynamic aspect. Further research, however, is needed to address the problem of handling highly cross-cutting aspects at runtime, although approaches such as composition filters [Bergmans and Aksit 2000] and the fragmented objects of AspectIX [Hauck et al. 1998] may be a step towards a solution.

Finally, software architectures [Shaw et al. 1995] have been proposed as an abstraction tool to model the structure of a system, in terms of configurations of components and the rules or styles governing such configurations. Although the technique has traditionally been used at design time, a number of researchers have pointed out the benefits of explicit runtime architectural representations, as a means to favour adaptation [Oreizy and Taylor 1998; Blair et al. 2000b]. In particular, software
architectures can be used as part of the self-representation of a system. This would enable reasoning about a system at the level of its whole configuration, in order to validate adaptations and preserve system integrity, such as proposed in [Moreira et al. 2001]. Another example is [Cazzola et al. 1999], where an architectural meta-layer representing configurations and strategies (rules governing the occurrence of interactions) is proposed as a means to achieve structural and behavioural reflection.

2.8 Summary

This chapter has presented the principles of reflective systems and illustrated their use in traditional areas, such as programming languages and operating systems. The application of these principles in middleware has also been considered, enabling an interpretation of what reflection means in this context. Current and proposed middleware architectures were analysed under the light of this interpretation, showing the major limitations of existing technologies and the way different research efforts have tried to solve these limitations using reflection at a more fundamental level.

As seen in section 2.5, existing middleware technologies and standards provide very limited support for platform flexibility. Such support is usually restricted to high-level services, while the underlying platform engine (the ORB) is largely considered a black box. Interestingly, though, recent developments in such technologies, such as the use of interceptors and policies, are a clear evidence of a trend towards more flexible middleware support. Nevertheless, the kind of flexibility provided is still limited to a few aspects of the platform, and is mainly restricted to initialisation time. This severely limits the use of middleware technology, especially in new application areas where requirements and environment can change at runtime, as is the case of mobile multimedia applications. Clearly, the provision of more comprehensive support for platform customisation and dynamic reconfiguration is a crucial requirement for the development of next-generation middleware platforms.

One promising approach to fulfil this requirement is represented by reflection and meta-level architectures, and several research efforts have been conducted in this direction. However, the existing approaches are usually restricted to a limited set of aspects of middleware implementations, also not considering many issues that are important in the design of reflective middleware, as discussed in section 2.6. The provision of a comprehensive reflection framework for middleware thus remains an
active research topic, and it is the goal of this thesis to contribute towards the definition of such a framework. Importantly, the requirements and design principles introduced in section 2.4 will be used as guidelines for the proposed approach.

Finally, is important to notice that other techniques (see section 2.7), also enable similar achievements. However, reflection is considered a more general approach, as it can be used as an overall framework that encompasses such techniques.
Chapter 3  Meta-Information Management

3.1 Introduction

Reflective techniques, as seen in the previous chapter, inherently deal with meta-information in order to build the self-representation of base-level entities. Meta-information is kept about the reified aspects of a system, in explicit or implicit form, as part of the state of the meta-level objects. Reflection, however, does not imply a consistent framework for the modelling and maintenance of this meta-information, especially considering issues of sharing and distribution. The provision of such a framework is precisely the goal of meta-information management techniques, and its presence, as discussed in Chapter 2, is an important, often overlooked requirement for reflective middleware. The aim of this chapter is to examine such techniques, setting the background for their use as a way to enhance the expressiveness of reflection. The concept of meta-information management is considered from the perspective of meta-modelling techniques, first introduced to help manage meta-data in database systems, but now widely recognised as an important concept in other areas, such as middleware.

In section 3.2, the basic definition and concepts of meta-information management are reviewed. This is followed by section 3.3, which describes the principal features of a meta-information management framework and the requirements that must be considered in the design of such a facility for middleware. Section 3.4 in turn presents a survey of the technologies and standards that play a fundamental role in the approach for meta-information management proposed in the thesis. Other related technologies are then examined in section 3.5, in order to give a broader view of the application of meta-information management. Finally, section 3.6 presents a summary of the issues raised in the chapter, along with considerations for the use of meta-information management from the perspective of this thesis.
3.2 Concepts and applications

3.2.1 Meta-information

The term *meta-information* is often used as a synonym for *meta-data*, which has its origins in the database community with the notion of self-describing databases. In that context, machine-readable meta-data is used to provide a description of the database itself, in terms of its schema. Thus, instead of describing the application domain, meta-data describes other data, especially regarding structural aspects. The common motivation is to enable the automatic interpretation of arbitrary data collections and the use of generic tools for their manipulation (e.g., data mining and browsing), possibly in a different context from the one in which the data were defined [Foote and Yoder 1998]. This generic use of meta-data is facilitated by techniques for meta-data management, which support the creation, storage and use of meta-data elements in an application-independent way [Mark and Roussopoulos 1986].

For the purposes of this thesis, however, the term meta-information is preferred instead of meta-data, due to the multitude of meanings that the latter receives. According to the definition adopted here, whilst meta-data is used to describe data in a general sense, meta-information is specifically used to describe the structure and semantics of first-class entities in a system (a similar definition is used in the MCAT meta-information catalogue [NPACI 1998]). In this way, a system can be augmented with meta-information describing the entities that were used in its composition, along with the relationships among them. Consequently, meta-information can be used as a way to represent the configuration of a system and each of its individual entities. Importantly, the effective use of meta-information requires a well-defined and uniform framework for its representation and use. Such a facility is commonly known as *meta-information management*, and is described in sub-section 3.2.3 below.

3.2.2 Related concepts

Modelling and Meta-modelling

The definition of meta-information is closely related to the concepts of *modelling* and *meta-modelling*. The former refers to the description of structured meta-information representing the runtime entities that compose a given target application
or system, as well as the relationships between such entities. The collection of such meta-information comprises a model of the application or system, and may provide sufficient detail to enable instantiation and introspection. On the other hand, meta-modelling refers to the ability to represent and manipulate meta-models, which are models that represent other models, thus defining the constructs used when modelling a system or application [Mili et al. 1995; Odell 1995]. As an example, consider the object-oriented paradigm, where applications and systems are modelled in terms of constructs such as classes and methods, as well as relationships among these constructs, such as inheritance. In this context, the role of the meta-model is to define the features that characterise the modelling constructs. This enables the meta-model to provide the necessary meta-information for model instantiation and introspection, in the same way as models are used to introspect on application entities.

Conceptually, therefore, the relationship between a meta-model and the models derived from it is one of instantiation, where the elements of a model are instances of entities defined in the meta-model. The meta-model can thus be seen as an abstract language (i.e., without necessarily having a concrete syntax) for describing models.

Importantly, the use of an explicit representation for meta-models (in terms of a meta-modelling architecture, as seen below) enables modelling constructs to be manipulated as first-class entities. Examples of this include object-oriented modelling techniques, such as the Universal Modelling Language (UML) [OMG 2000d], which has its meta-model defined using the MOF (see section 3.4.2) meta-modelling architecture. Similarly, several middleware technologies, including those considered in the previous chapter, can be represented in terms of explicit meta-models, which define their respective programming models. For instance, the OMG object model corresponds to the CORBA meta-model, which is represented using the concrete syntax of OMG-IDL. In a similar way, RM-ODP can be described as a set of inter-related meta-models corresponding to its five viewpoint languages.

Meta-modelling architectures

A meta-modelling architecture generalises the above discussion on models and meta-models, and usually consists of distinct levels [Odell 1995; Mili and Pachet 2000], as described in Table 3.1 (although level-independent meta-modelling has also been advocated [Atkinson and Kuhne 2000]).
### Table 3.1 – Levels of a meta-modelling architecture

<table>
<thead>
<tr>
<th>Level</th>
<th>Description and examples</th>
<th>Typical constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta-…-meta-model</td>
<td>definition of the language used for describing the level underneath</td>
<td>appropriate meta-…-modelling constructs to describe the entities of the level underneath</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>meta-meta-model</td>
<td>definition of the meta-modelling language, in terms of the constructs used to describe and interpret the structure and semantics of meta-models – e.g., MOF, CDIF, OIM</td>
<td>meta-meta-meta-information, e.g., entity, association, module and semantic constraints</td>
</tr>
<tr>
<td>meta-model</td>
<td>definition of the modelling language, in terms of the constructs that can be used in the definition of models – e.g., meta-models associated with CORBA, COM, Java RMI, RM-ODP and UML</td>
<td>meta-meta-information describing what comprises the definition of a model element, e.g., class, interface, method and attribute</td>
</tr>
<tr>
<td>model</td>
<td>definition of the structure and semantics of applications, in terms of the types of the entities that compose the application – e.g., the model of a banking application</td>
<td>meta-information representing particular class, interface, method or attribute definitions; data types; relationships, etc</td>
</tr>
<tr>
<td>information level</td>
<td>instances of model elements – e.g., the runtime entities of the modelled applications and systems</td>
<td>objects, methods, attributes, data values, references, etc</td>
</tr>
</tbody>
</table>

In particular, the *instantiation* relationship between the entities of models and meta-models can be generalised and applied to any pair of adjacent levels. Consequently, the existence of a level enables the concepts of the level below it to be treated as first-class entities. According to [Atkinson 1997], however, the meaning of the instantiation relationship depends on the actual approach to meta-modelling. In *strict meta-modelling*, every entity at a given level must be an instance of one and only one entity at the level immediately above. With *loose meta-modelling*, in contrast, an entity is permitted to be an instance of another entity at the same level. Implications of the two approaches will be considered in Chapter 5. In this thesis, strict meta-modelling is adopted, meaning that a given level can be seen as either consisting of models for the level below or consisting of entities that are instances of model elements from the level above. This means that the term “meta” should be used in a relative sense. In the case of the topmost level, however, its definition is typically achieved in a meta-circular way, with its constructs used for their own definition. This enables such constructs to be treated explicitly and avoids the need for infinite levels of meta-modelling. Therefore, only a small number of levels, usually four, are required, as seen in the standards examined in section 3.4. Importantly, the topmost level should provide a single, unified meta-modelling language.
3.2.3 Management of meta-information

The use of a meta-modelling architecture is the basic prerequisite for meta-information management. The levels of meta-modelling (Table 3.1) provide the overall organisation for maintaining and using meta-information and also enable an open-ended approach, where, for instance, meta-models can be extended using the meta-meta-model constructs. However, in order to have an effective architecture for the management of meta-information, it is also necessary to provide facilities which, on top of the meta-modelling architecture, assist with [Crawley et al. 1997a]:

- **meta-information definition**, such as with a language with well-defined syntax and semantics (conforming to the meta-model), as well as tools, such as compilers to validate and translate textual meta-information into a machine-readable form; alternatively, interactive tools (such as with a GUI) can be used for this purpose;

- **meta-information maintenance**, with a distributed and persistent repository with features for creating, deleting, managing and manipulating meta-information;

- **definition, storage and evaluation of relationships**, such as compatibility and substitutability, between different entities of meta-information; and

- **meta-information interchange**, based on mappings and tools to transfer meta-information between different repositories, possibly using different meta-models.

Importantly, the use of a unified architecture for meta-information management enables a consistent view of meta-information in a distributed environment. This is crucial when meta-information must be shared by different users in a heterogeneous setting, as conformance to a common meta-model enables its unambiguous interpretation [Indulska et al. 1993]. Furthermore, typical meta-information management architectures impose the requirement that meta-information, once defined and published, must be **immutable**, so that replicated meta-information elements are always kept consistent [ITU-T/ISO 2000]. This is also important, especially in open distributed systems, as meta-information is used to provide a reliable description of runtime objects (e.g., to enable dynamic type checking).

3.2.4 Relationship to reflection

Due to their ability to support the meta-representation of a system (through the model used in its instantiation), meta-information management can be considered as a
form of *intensional reflection*, as described in the previous chapter. Although usually limited to introspection, such techniques enable arbitrary aspects of a system to be represented, in terms of explicit model elements. Not surprisingly therefore, meta-modelling techniques can be compared to the meta-class approach to reflection, as the relationships among instances, models and meta-models are of the same nature as those among objects, classes and meta-classes. Indeed, languages with meta-class support are one of the ways to implement a meta-modelling facility [Mili et al. 1995].

However, even considering the meta-class approach, reflection and meta-information management are essentially different techniques. In particular, reflection implies a causal-connection link between entities that exist at different levels, such as between classes and their meta-classes. This is not the case with meta-information management, due to the usual immutability of meta-information. Furthermore, if the *specific meta-object approach* to reflection is considered, particular instances have their own meta-representations, which can evolve independently from those of other instances. This contrasts with the many-to-one relationship between instances and their respective modelling entities in meta-information management.

Nevertheless, instead of excluding each other, these two techniques are potentially complementary, as suggested in [Crawley et al. 1997b]. Meta-information management works as a *top-down* technique, as the meta-level entities exist *a priori* and are used to create the base-level entities of a system. This means that meta-information management characterises a generative approach, as several base-level entities can be created from the same meta-representation, thus sharing a common description at the meta-level. On the other hand, reflection takes a *bottom-up* approach to meta-representation, in which the base-level entities are the focus of interest, and their existence is typically independent of the actual existence of the meta-level. More specifically, the relationship between base- and meta-level entities in object-orientated reflection is one of reification, and works at an *extensional* level, as opposed to the instantiation relationship used in meta-information management, which is intensional in its nature. Interestingly, these distinctions suggest that the two techniques can be used in conjunction, in order to provide different ways to view, use and manage the meta-representation of a system. This relationship will be explored as one of the central objectives of this thesis, as seen in Chapter 5.
3.2.5 Applications

According to [Mili et al. 1995; Mili and Pachet 2000], the need for meta-modelling and meta-information management techniques arises in three major areas:

- **domain knowledge abstraction**, to factor out and manage, at the meta-model level, knowledge shared among different applications in a specific domain, enabling the generation of particular application models by refining the meta-model;
- **behaviour representation**, to explicitly represent computational entities (such as messages, contexts, and scheduling), their properties and the ways they behave;
- **representation of modelling languages**, to allow the definition of the meta-models used in the description of application models, as well as to assist in the translation among such meta-models (by using the meta-meta-model as a logical bridge).

This thesis is concerned with the use of meta-information management and related techniques for the representation of modelling languages (although the issues related to meta-model translation are regarded as future work). In particular, the thesis considers modelling languages in the context of middleware, where they can be used to describe the types of platform components, along with their respective structure and configuration. Notably, the modelling languages of interest follow a well-defined programming model, with an underlying *type system*. In this context, meta-information management techniques become the equivalent of the functions of *type management* and *configuration management*, which are discussed in the next section.

For completeness of the discussion, other concrete areas where meta-information management and meta-modelling techniques can be used for the representation of modelling languages are briefly considered in Table 3.2.

Regarding the other two major areas mentioned above, however, a different perspective is adopted in this thesis. Behaviour representation is considered as the realm of reflection, although it could be argued that the use of meta-information management would enable a unified framework for maintaining behavioural meta-information. Similarly, the use of meta-modelling for domain knowledge representation will not be considered, as it can be achieved with conventional object-oriented techniques, such as generalisation, as discussed in [Atkinson et al. 2000].
Table 3.2 – Areas where meta-modelling and meta-information management can be applied, in the context of the representation of modelling languages

<table>
<thead>
<tr>
<th>Area</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database management systems</td>
<td>System databases (or data dictionaries) are employed to represent the schema of application-specific databases [Mark and Roussopoulos 1986].</td>
</tr>
<tr>
<td>Meta-computing</td>
<td>A repository of component types and strategies for their use are employed to optimise the configuration of high-performance distributed computations, such as in [Fitzgerald et al. 1997].</td>
</tr>
<tr>
<td>Multimedia document management and modelling</td>
<td>Meta-information is extensively used for the description, classification, location, and interpretation of the content of multimedia streams and documents, such as in [Bohm and Rakow 1994], [Lougher et al. 1997], and in the proposed multimedia standard MPEG-7 [ISO 2001].</td>
</tr>
<tr>
<td>WWW meta-information frameworks</td>
<td>Meta-information is used to describe the content, properties and semantics of web-related resources, facilitating meaningful classification and search, such as is the W3C Resource Description Framework (RDF) [W3C 1999].</td>
</tr>
<tr>
<td>Digital libraries</td>
<td>Meta-information facilitates access to structured collections of information and enables enhanced modes of access to such libraries [Smith 1996].</td>
</tr>
<tr>
<td>Data warehousing</td>
<td>Meta-information management is crucial for the organisation of enterprise data, in order to facilitate the maintenance, classification, mining and analysis of such data, for instance, as defined in the OMG Common Warehousing Metamodel (CWM) standard [OMG 2000b].</td>
</tr>
<tr>
<td>Model-driven engineering</td>
<td>Complete systems can be generated from implementation-independent models, by refining such models with application- and platform-specific choices [Dirckze and Iyengar 2000]. This approach has been adopted by OMG in the proposed Model Driven Architecture (MDA) [Soley 2000].</td>
</tr>
<tr>
<td>Active object models</td>
<td>An explicit, causally connected representation of the model of a system enables runtime adaptation [Foote and Yoder 1998].¹</td>
</tr>
</tbody>
</table>

3.3 Meta-information management for middleware

3.3.1 Basic requirements

The availability of runtime meta-information is essential for the maintenance of networked and distributed systems, such as in the case of network management protocols, which enable the gathering and analysis of meta-information related to network performance. In middleware, similar needs arise, such as when the several components of a platform need to be continuously monitored in order to guarantee the agreed levels of quality of service. Besides this basic use, however, middleware platforms present further requirements for a more comprehensive meta-information management facility [Kutvonen 1997]. In particular, middleware platforms need to support an open services environment, where service users dynamically bind to service providers. In addition, in such an environment, new services can be

¹ Although similar to object-oriented reflection, this approach works at an intensional level instead.
dynamically introduced and existing services can evolve through versioning. This demands runtime meta-information that describes the types of the currently available services and the requirements of service users (e.g., in terms of the signatures and semantics of their interfaces). Such meta-information is vital for the dynamic discovery of services, as well as for the type checking of service types before binding and for the bridging of service interfaces that are semantically compatible but structurally incompatible. Meta-information management techniques thus play an important role in open distributed systems, as a means to organise and maintain such runtime meta-information and to support service discovery and interoperability.

Importantly, in order to enable the use of meta-information management techniques, it is essential that middleware platforms be defined around a precise meta-modelling architecture. The existence of an unambiguous meta-model (cf. type system) defining the kinds of artefacts used in middleware configurations, in terms of their structure, semantics and relationships, is crucial to enable meta-information to be organised in a meaningful way. Also important is the use of a unified meta-meta-model, which enables the description and interpretation of the different meta-models that can exist in an environment where multiple middleware technologies need to interwork. In this context, the meta-meta-model enables the translation of equivalent concepts between different middleware meta-models. This resembles the concept of a universal meta-information facility (or repository), as defined in [Iyengar 1998].

In the remainder of this section, important applications of meta-information management for middleware are considered in more detail, with emphasis on those that are most closely related to the approach presented in the thesis. In particular, type and configuration management are examined, along with their fundamental underlying concepts. For completeness, other applications of meta-information in middleware are briefly considered at the end of the section.

3.3.2 Type and configuration management

Types, templates, relationships and type systems

The concepts of type and configuration management are a refinement of the concept of meta-information management where the kinds of meta-information of interest consist of descriptions of the entities used in the construction of systems. In
particular, this thesis is concerned with the application of these two concepts in the
construction of middleware platforms. The underlying terminology and concepts are
adopted from RM-ODP [ITU-T/ISO 1996] and are defined below.

A *type* is defined as a predicate that characterises a collection of entities, in such a
way that evaluating the predicate over a given entity can determine whether it is an
instance of the given type. A type can be associated with one or more *templates*, each
one specifying enough details to allow the creation of instances of the type in a
particular environment or using different implementation strategies. Interestingly,
however, a type can be considered a template in environments where the predicate is
sufficient for the instantiation process to take place [ITU-T/ISO 2000]. Conversely, a
template (together with its underlying predicative type) can be considered as a type
which is detailed enough to be used for instantiation in a given environment or
platform [Kutvonen 1997].

While templates are primarily used for instantiation (as they describe internal
implementation details), types are mainly used to describe the properties of
computational entities, in terms of their externally visible behaviour (e.g., in terms of
interface signatures). Notably, *relationships* can be defined between types, in a way
that enables meaningful comparisons between their instances. For instance, a type
may be considered a *subtype* of another one (called a *supertype*), in which case the
predicate for the latter also holds for all instances of the former. In practice, this
means that instances of the subtype can be used to replace instances of the supertype
in a given system configuration. Another useful kind of relationship is *type
compatibility*, used to compare (type-check) the types of instances for the purpose of
interoperation. Such relationships between types can be stored along with the type
definitions or, alternatively, they can be computed at runtime.

In the context of a meta-modelling architecture (see Table 3.1), types, relationships
and templates are considered as elements of the model level, since they describe the
running entities of a system. The meta-model level, in turn, is populated by *meta-types*
and *relationship types*, which define the available kinds of types, templates and
relationships. A meta-model can thus be used to define a *type system*, which consists
of a closed (i.e., self-contained) set of meta-types and relationship types that defines
the terminology and the programming model used to describe systems in a particular
domain or platform [Crawley et al. 1997b].
The modelling of types, templates and relationships as instances of meta-model elements enables them to be treated as first-class entities, which can be dynamically manipulated at runtime (e.g., for introspection purposes). In a similar way, the existence of a further level of modelling (the meta-meta-model) gives first-class status to the meta-model elements that comprise type system descriptions, with two important consequences. First, existing type systems can be dynamically extended to incorporate new meta-types. Second, several type systems can be managed within the same framework, opening the possibility for bridging (i.e., translating) similar concepts across different type systems. These two aspects, however, will not be further explored in this thesis, and are considered as possibilities for future work.

Type management

Type management can be defined as the ability to associate unambiguous names to type specifications and to maintain relationships between such names, in order to support dynamic type matching and type checking [Brookes et al. 1997]. Typically, the following functions are provided by a type management facility:

- support for the definition and creation of new types;
- explicit support for the definition and evaluation of relationships among types; and
- maintenance of a persistent repository of type and relationship definitions, with dynamic support for search, retrieval and comparison of types.

In addition, support for the integration of software engineering tools is typically among the goals of a type management facility, as the type-related concepts can be used to bridge the different phases of the software lifecycle [Brookes et al. 1997].

Examples of type management architectures include the work reported in [Brookes and Indulska 1994] and [Brookes et al. 1997], and also [Christiansen et al. 1997]. In common, these papers target the management of service types in open distributed systems by adopting a modular architecture for the type manager, with distinct modules for each of the type management functions (such as storage, query parsing, relationship computation and version management). However, they adopt a flat view towards the representation of meta-information, without explicit meta-modelling levels. Thus, the management of meta-meta-information needs to be accommodated at the same level as meta-information, resulting in the ad-hoc representation of a limited
range of meta-meta-information elements (such as relationship types and read-only meta-types). Furthermore, the lack of an explicit meta-meta-model precludes any extensions to existing type systems (apart from the ability to introduce new relationship types as reported in [Brookes et al. 1997]).

In addition, other more advanced type management functions can be defined, notably related to the concept of bridging different type systems (as mentioned above) and to the notion of type evolution. Examples of architectures with such functionality are, respectively, [Peltier et al. 2000], where a generic formalism for model transformation is developed which enables a model defined according to a given meta-model to be translated into another meta-model, and [Senivongse and Utting 1996], where type versioning is employed to evolve service types in a distributed system and mapping functions are defined to determine compatibility between type versions. Crucially, the use of a meta-information architecture of some sort (such as the ones examined in section 3.4) is essential to effectively enable such functionality.

Configuration management

Another area where the management of meta-information can prove useful is in the configuration of open distributed systems [Crane et al. 1995]. In this context, the configuration activities include object creation and allocation (to physical locations in the system), and binding of objects through their interfaces. Traditionally, as in [Fossa and Sloman 1997], such activities are carried out by interpreting textual or graphical specifications of the structure of the software objects comprising a distributed application or service. These specifications can be used, for example, to describe the internal configuration of a composite component, with sufficient detail to allow its instantiation. In addition, such specifications serve as runtime documentation of the configuration of a system and its components, providing a basis for reconfiguration.

As configuration specification constitutes meta-information about a system, techniques of meta-information management can be used to assist the whole process of system configuration and reconfiguration. Firstly, meta-information management enables the persistent storage of configuration specifications in the form of templates (see section 3.3.2) in a managed repository. Secondly, the association between templates and types enables the use of type relationships to search for and compare configurations, as well as to validate interconnections between the elements of a
configuration. Thirdly, *version management* can be used to control and track the evolution of configurations, by enabling multiple template versions to co-exist. Finally, as templates are regarded as entities of the model level (see Table 3.1), the language used for system configuration can be defined as part of a meta-information management framework (at the meta-model level). Such language defines the kinds of templates used in the specification of configurations. In addition, the use of a meta-meta-model level makes possible the integration of different such configuration languages (e.g., by translating templates from one language into another).

### 3.3.3 Other applications in middleware

Although type and configuration management constitute the main applications of meta-information management from the point of view of this thesis, other important applications of the concept can be devised in the context of distributed systems middleware. Notably, runtime meta-information is essential to support the advertisement and discovery of service providers based on high level descriptions of their types and properties, as prescribed by the RM-ODP trading function [ITU-T/ISO 1997]. In this context, type management plays an important role, as type meta-information is required to validate and match service provisions [Indulska et al. 1993]. Note, however, that the trader itself acts as a meta-information repository, as it needs to maintain meta-information that is specific to particular instances of a service.

Other applications of a meta-information management facility in middleware include the provision of runtime interface descriptions, in order to support the dynamic assembly of method requests, as well as the provision of meta-information to support infrastructure functions (such as naming, location and migration).

### 3.4 Key standards

#### 3.4.1 The CORBA Interface Repository

**Overview**

The Interface Repository (IR) is the central component of the CORBA architecture for managing meta-information defined according to the CORBA meta-model (or type system). The description of the CORBA IR presented here corresponds to its
specification as in the proposed CORBA 3.0 release of the standard [OMG 1999a]. As such, the CORBA IR provides for the persistent storage and management of type meta-information defined in OMG IDL, such as for interfaces, components, structs, unions, enums, arrays and sequences. The IR also provides for the representation of primitive types (by means of type codes), as well as for the storage of non-type meta-information, such as constant, operation, attribute and exception definitions.

Meta-modelling architecture

Meta-information in the IR is maintained as a set of CORBA objects, which are instances of meta-types that represent each of the kinds of meta-information supported by the repository (such as the ones mentioned above). The meta-types in turn are also defined as CORBA objects, with IDL interfaces for access to the meta-meta-information they represent. This means that the CORBA IR is conceptually defined in a meta-circular way, as the meta-model objects (meta-types) are defined using the same constructs that are used to define application model objects (types). This is illustrated in Figure 3.1(a). As an example, the meta-type for interface definitions in (InterfaceDef) is itself specified in terms of an interface definition.

In principle, this meta-circular definition means that the CORBA IR meta-model can be dynamically extended to accommodate new meta-types, with the use of its own features. However, it is not possible to represent the semantics of newly introduced meta-types, notably constraints characterising their consistent use. Such constraints are defined as part of the CORBA IR specification but, because the IR concentrates on the modelling of structure rather than semantics, their definition is typically hard-coded in the repository implementation. This limits the extent to which the CORBA IR can be used to extend itself (for instance, the IR does not offer support for tools that would enable automatic generation of repository code based on new meta-type definitions). Nonetheless, this kind of use is not one of the goals of the CORBA IR. Instead, the goal of the repository’s self-definition is simply to allow access to meta-information (instances of meta-types) in the form of normal CORBA objects, which can be accessed in a location-independent way via the ORB mechanisms. This intent (and its realisation in practice) is better captured in Figure 3.1(b), which shows the definition of the IR meta-model as just one more model, along with normal application models. Therefore, the meta-model represented by the CORBA IR should
be viewed as a fixed meta-model, while meta-model extensibility is considered an issue for the OMG meta-modelling standards (as described in subsection 3.4.2). In what follows, the major meta-types of this meta-model are described, with emphasis on the structure used by the CORBA IR for their representation.

![Figure 3.1 – Self-definition of the CORBA IR: (a) conceptual; (b) actual](image)

**Structure of the CORBA IR**

The structure in which meta-information is stored in the Interface Repository mirrors the hierarchical scoping of OMG IDL definitions. Individual IDL definitions are stored as *interface repository objects* (IR objects, for short), which may contain or be contained in other IR objects, thus forming a containment hierarchy. The root of this hierarchy is represented by a container of kind `Repository` (or `ComponentRepository`, if component definitions are supported). The IR objects contained in the repository may be organised into separate modules (IR objects of kind `ModuleDef`). In addition, other kinds of IR objects can also be containers, as is the case of `InterfaceDef` instances, which can contain, e.g., `OperationDef` and `AttributeDef` IR objects. The overall structure of IR is illustrated in Figure 3.2.

Each of the meta-types shown in the figure has a standard IDL interface, with attributes to represent their properties, as well as operations for their manipulation and for navigation of the repository contents. Instances of these meta-types are thus normal CORBA objects, meaning that individual IR objects can be directly accessed through their interfaces, and that IR objects usually reference each other via CORBA object references. Consequently, the actual repository structure can be physically distributed (by partitioning or replicating the IR objects across different nodes in a
distributed system). In addition, individual IR objects (except for the anonymous ones) have globally unique repository identifiers, which allow a given type to be named unambiguously, even when its corresponding IR object is replicated.

![Repository (or ComponentRepository)](image)

Figure 3.2 – Containment hierarchy of the CORBA 3.0 Interface Repository.

Navigation through the interface repository, as seen above, is provided by operations and attributes at the interfaces of IR objects, which enable the links between container and contained objects to be followed. For instance, the defined_in attribute of an IR object is a reference to the object’s immediate container, while the contents operation of a container IR object enables the enumeration of the objects defined within the container. In addition, special lookup operations available at the interface of container IR objects enable searching through the repository contents. For example, the lookup operation enables an IR object to be located relatively to the container upon which it was invoked, while the lookup_id operation allows an IR object to be located based on its unique repository identifier.

---

2 The definition of each of the meta-types shown in the figure can be found in the CORBA 3.0 draft specification [OMG 1999a]. Note that other meta-types exist, which correspond to anonymous IR objects, not shown in the figure. Also, note that the ComponentDef and HomeDef meta-types (and their accessory meta-types) are only present if the root of the hierarchy is a ComponentRepository.
Other type management features of the CORBA IR

Besides the basic ability to manage a collection of type definitions and to allow searching and navigation through this collection, the IR also provides other useful type management functions. Basic facilities include runtime type-checking of request signatures (e.g., for use with dynamic invocations) and conformance checking for interface types (to determine interface substitutability). In the case of interface types, type checking is performed by direct comparison, based on the repository identifiers of the respective IR objects. However, in the case of non-interface types (e.g., structs), the IR uses type information in the form of TypeCode objects, which enable comparison based on the complete structure of the type.

In addition, the IR also provides support for other components of an ORB, as well as for tools that need to manipulate type meta-information. This includes:

- support for connecting different ORBs, by replicating the IR objects that represent the shared types (as all replicas of the same type definition will have the same repository identifier, inter-ORB requests can be unambiguously interpreted);
- provision of interface meta-information to IDL compilers (to enable consistency checks of the inheritance graph when adding new interface definitions);
- support for CASE tools, such as browsers of meta-information;
- support for factories in the instantiation of components and their interfaces; and
- basic support for version control of type definitions, in the form of version numbers associated with every IR object (though the IR itself does not specify how these version numbers should be managed).

3.4.2 The Meta-Object Facility and related standards

Principles

The Meta-Object Facility (MOF) specification [OMG 2000c] forms the centre of the meta-information strategy of the OMG. In comparison to the Interface Repository, the MOF adds value in terms of a generic architecture for meta-information management. Notably, the MOF adopts a four-level meta-modelling architecture, enabling the definition and management of models and meta-models. It also defines the foundations for meta-information management in this architecture, in terms of
repositories with standard IDL-defined interfaces that allow runtime access to meta- and meta-meta-information. Importantly, the MOF deals with meta-information management from an open-ended perspective, providing support for model and meta-model evolution, as well as for the dynamic assimilation and integration of meta-models (based on the runtime introspection and interpretation of their descriptions). Two kinds of users are envisaged for the MOF. First, it can be used by model engineers for the definition of models and meta-models that represent meta-information in specific domains. Second, it can be used by software developers in order to discover (introspect on) the structure and semantics of the resources needed in the construction of systems.

The original aim of the MOF was the management of meta-models to assist in the analysis and design of software systems. In the context of the OMG, this means that the MOF has become the unifying point for integrating the several different meta-models that are used in the software development process. The notable examples are the CORBA meta-model (symbolised by OMG-IDL) and the Unified Modelling Language (UML) [OMG 2000d]. However, the scope of the MOF has been extended, so that MOF-based meta-models can be defined for arbitrary application domains. Well-known examples are data warehousing and information management (e.g., to describe the content and structure of WWW resources). In general, the only requirement for the MOF to be applicable in a given domain is that some structure can be imposed on the meta-information to be represented.

Meta-modelling architecture

The MOF follows the principles of the abstract meta-modelling architecture discussed in section 3.2.2. This means that its architecture is organised in layers, as illustrated in Figure 3.3, where entities at a given layer are defined as instances of entities at the layer immediately above, according to a strict meta-modelling approach [Atkinson 1997]. The top layer, M3, is the meta-meta-model level, which is fixed and corresponds to the standard MOF Model (see below), the language used to define meta-models in the MOF. Importantly, layer M3 is defined in a meta-circular way, meaning that the MOF Model is described using its own constructs. The next layer, M2, is populated by the meta-models defined using the MOF model. Each meta-model can be seen as offering a notation for the modelling of meta-information in a
particular domain and according to specific requirements or viewpoints within that domain. Examples are the standard OMG meta-models, such as UML (for object-oriented analysis and design), IDL (for interface descriptions), and CWM (for the description of data warehouse models). Importantly, however, layer M2 is not restricted to the standard meta-models, as new meta-models can be defined and existing ones can be extended, in order to suit the requirements for meta-information management in specific contexts.

![Figure 3.3 – Example of use of the MOF meta-modelling architecture](image)

Layer M1 then is the actual meta-information layer and consists of models representing specific applications or systems. Each model is defined according to a single M2-level meta-model, which provides the necessary meta-meta-information to instantiate and to interpret the model elements.

Finally, layer M0 deals with the entities that are the ultimate target of modelling, such as the runtime objects of a system. The role of this layer is controversial, as some interpretations (including the one in the preface of the MOF specification [OMG 2000c]) consider application programs as residing at this layer. In this thesis, however, the interpretation is that layer M0 consists of the entities that strictly result from the instantiation of M1-level model elements. Application programs, which may define models, are thus considered at level M1. Note, however that layer M0 is not treated explicitly in the MOF. Nevertheless, this layer is crucial in the broader context of this thesis, as its entities are a prime concern in the proposed reflection framework.
The MOF Model

The meta-meta-model of the MOF (the *MOF Model*, for short) consists of a simple (though not minimal) set of object-oriented constructs that are generic enough to allow the description of arbitrary meta-models. These constructs constitute the meta-meta-types of the MOF, which are used to instantiate meta-model elements (meta-types) at layer M2. The main constructs of the MOF Model are described in Table 3.3 below.

**Table 3.3 – Major constructs of the MOF Model**

<table>
<thead>
<tr>
<th>Meta-type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>This meta-meta-type is used to instantiate class meta-types, which represent the first-class elements of meta-models. A class meta-type may include attributes and operations to represent the state and the functionality of MOF-defined type objects. A class meta-type may also contain references to other class meta-types, as well as constants, data types, exceptions and other meta-modelling elements. Examples of concepts that can be represented as class meta-types include interface, operation and attributes (such as in the CORBA meta-model). Importantly, class meta-types can also be defined through inheritance, thus enabling the incremental development of meta-models.</td>
</tr>
<tr>
<td><strong>Association</strong></td>
<td>Instances of this meta-meta-type represent binary relationships between the first-class elements of a meta-model (i.e., class meta-types). Examples of association meta-types are the inheritance relationship between interfaces, and the containment relationship between interfaces and operations (again, such as in the CORBA meta-model).</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>This construct of the MOF Model provides a convenient way to access associations, in such a way that an endpoint of an association can read or modify the other endpoint using attribute-like access operations (via <code>get</code> and <code>set</code> accessor methods).</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>This construct of the MOF Model provides for the modularity of meta-model descriptions, by grouping related meta-model elements. Typically, a whole meta-model is enclosed within a package, although a meta-model can also be partitioned into several packages to improve clarity. A package can thus contain instances of any of the constructs defined in the MOF Model, including nested packages. In addition, a package can reuse other packages via inheritance or importation.</td>
</tr>
<tr>
<td><strong>Data type</strong></td>
<td>This meta-modelling concept is used to represent anonymous types (which do not correspond to identifiable elements of a meta-model). Instead, data types are used as part of the specification of other meta-model elements, such as to give the types of class attributes and operation parameters. Currently, the MOF Model supports the set of non-interface data types specified for OMG-IDL.</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td>This meta-modelling construct enables meta-model definitions to be augmented with semantics, in the form of rules that define the consistency of the models derived from a meta-model. In principle, constraints can be expressed in any language, including a natural language (in which case they have to be interpreted manually). However, the MOF specification recommends the use of a formal language, the Object Constraint Language (OCL), which is defined as part of the UML specification [OMG 2000d]. Typically, OCL constraints take the form of invariant conditions that must hold for all instances of a meta-model, and which can be interpreted by modelling tools in an unambiguous way. In addition, the use of OCL enables tools to automatically generate repository code that conforms to the meta-model constraints.</td>
</tr>
</tbody>
</table>
Computational model

As the MOF is primarily intended for the management of meta-information within the OMG Architecture, its services must be made accessible through the CORBA ORB. To this end, the MOF is structured in terms of CORBA objects, with IDL-defined interfaces. These objects typically represent meta-types and types, and are called \((\text{meta-})\text{meta-objects}\) in the context of the MOF, although in this thesis they will be referred to as \(\text{MOF (meta-)}\text{meta-objects}\), in order to avoid confusion with reflection terminology. A set of related MOF meta-objects, usually structured as a graph, constitute a MOF-based repository facility, which can be augmented with tools to facilitate the access to meta-information. Note, however, that MOF services can also be accessed using protocols based on XML [W3C 2000], as described below, meaning that the use of a MOF-based facility is not limited to CORBA environments. Figure 3.4 illustrates the positioning of the MOF within the OMG and XML environments.

![Figure 3.4 – The MOF and its relation to the OMG architecture and to XML](image)

The OMG-IDL interfaces of MOF meta-objects are divided into two categories: meta-model specific interfaces and generic (meta-model independent) interfaces.

The \text{meta-model specific interfaces} are defined to mirror their respective meta-models, by applying, to the meta-model descriptions, a standard mapping of the MOF Model concepts into OMG-IDL, the \text{MOF-to-IDL mapping}. The major concepts of the MOF Model are mapped into IDL as summarised in Table 3.4 below. In order to facilitate the creation of meta-model specific interfaces, the MOF includes a set of templates for IDL generation. Note that a mapping is also defined for the auxiliary
concepts of the MOF Model, such as operations (which are directly mapped onto IDL operations), attributes (mapped onto operations to get and set the attribute values), and references (mapped onto operations to access and modify reference values).

Table 3.4 – Main elements of the MOF-to-IDL mapping

<table>
<thead>
<tr>
<th>MOF Model construct</th>
<th>Mapping to OMG-IDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>A package defined in the meta-model is mapped onto two IDL interfaces, a Package Factory interface and a Package instance interface. The former is used to create M1 level instances of the package (i.e., package meta-objects), while the latter represents such instances and contains attributes that represent the contents of the package.</td>
</tr>
<tr>
<td>Class</td>
<td>A class meta-type is also mapped onto two distinct IDL interfaces, a Class Proxy interface and a Class instance interface. The former serves as the factory and container for M1 level instances of the class meta-type (i.e., class meta-objects). Class instance interfaces in turn represent particular class meta-objects, and support access to the operations, attributes and references defined in the class meta-type.</td>
</tr>
<tr>
<td>Association</td>
<td>An association meta-type is mapped onto an IDL interface representing all instances of the association (i.e., the set of all links derived from the association) and containing operations for querying and updating the association instances.</td>
</tr>
</tbody>
</table>

The meta-model independent interfaces, on the other hand, are fixed by the standard, and are known as the MOF Reflective Interfaces. These are a group of four IDL interfaces: RefBaseObject (for traversal of MOF repositories and between the modelling layers), RefObject (the general-purpose equivalent of Class and Class Proxy interfaces), RefAssociation (for the manipulation of links between objects, independently of the association meta-types), and RefPackage (for the generic manipulation of the contents of a package). These interfaces allow access to meta-information in a way that is independent of the particular meta-model in use.

Finally, the meta-information represented by MOF meta-objects can also be accessed via the Extensible Markup Language (XML), using a standard MOF-to-XML mapping defined by the OMG, the XML Meta-data Interchange format (XMI) [OMG 2000e]. This kind of access to the MOF has two distinct purposes:

- to provide a standard transfer syntax for MOF-based meta-models, enabling the serialisation and transport of large amounts of meta-information among modelling tools (as the MOF-to-IDL mapping is not efficient for this purpose); and
- to enable tools and applications outside the CORBA environment to access MOF-based meta-information, taking advantage of the widespread use of XML.
In short, XMI defines the MOF-to-XML mapping in terms of a set of XML DTD (Document Type Definition) production rules, used to transform MOF-based meta-models into XML DTDs. These DTDs are then used to guide the encoding of MOF-based meta-information (i.e., models) in the form of XML documents.

Importantly, the MOF specification includes definitions of the MOF Model itself, in terms of IDL interfaces (by applying the MOF-to-IDL mapping) and as an XML DTD. This allows the MOF meta-meta-model definition to be used to interchange meta-models, similarly to the way meta-models are used to exchange models.

Implementing meta-models and repositories with the MOF

The MOF specification defines the process of model and meta-model management in an abstract way, based on the MOF mappings and on an abstract repository framework. Individual implementations of the MOF are thus free to define particular details of this process, in order to suit their intended applications. A typical realisation of the MOF is described below, based on DSTC’s dMOF [DSTC 2001]. A simplified view of the process of meta-model instantiation with dMOF is shown in Figure 3.5.

![Figure 3.5 – Model and meta-model management with the MOF](image-url)

The first step in the process invariably consists in the description of the target meta-model, using some modelling notation, such as the UML or the Meta-Object Definition Language (MODL) [DSTC 1997]. The meta-model description is then converted (by a compiler) into MOF meta-meta-objects, stored in the MOF repository.
server. From these meta-meta-objects, specific IDL interfaces can be generated for the MOF meta-objects, using the MOF-to-IDL mapping. In addition, code can be automatically generated to implement the meta-model objects, according to the standard semantics defined by the MOF-to-IDL mapping. These automatically generated implementations are called MOFlets. A specific meta-model repository can then be produced by compiling the IDL interfaces and their respective MOFlets. This repository offers facilities to create, access and update MOF meta-objects representing the elements of user-defined models. Applications and tools that need access to meta-information can then interact with the MOF meta-objects via the CORBA ORB. Note that the generated MOFlets can also have hooks that enable repositories to be customised with user-defined semantics. In addition, a MOF implementation can also use meta-model specific definitions to generate tools that facilitate access to meta-information in meta-model repositories, such as GUI browsers and editors.

### 3.4.3 The RM-ODP Type Repository Function

**Overview**

The Type Repository Function is one of the repository functions prescribed in the architecture reference model of RM-ODP [ITU-T/ISO 1995b], along with the Storage, Information Organisation, Relocation and Trading functions. The Type Repository Function manages a repository of type definitions and type relationships, and has an interface for each type specification it stores. It also includes facilities for creating types and their respective interfaces, as well as for manipulating individual types, such as: (a) querying the specification of the type; (b) automatic derivation of relationships between types, along with support for the introduction of relationships that cannot be automatically derived; and (c) querying relationships between types.

The Type Repository Function is essential to other ODP services and functions, such as binding and trading, and a precise definition of its architecture was the subject of recent standardisation effort [ITU-T/ISO 2000].

**Architecture**

Instead of defining a new type management framework, the ODP Type Repository Function is based on the concepts and architecture of the OMG Meta-Object Facility.
However, the specification of the Type Repository Function further refines the MOF framework by providing a context for its use in ODP systems. The specification is presented in terms of three of the ODP viewpoints, as described below.

**Enterprise specification.** This is the main addition in comparison with the MOF. The Type Repository Function is specified in terms of the objectives and policies that govern its activities. Explicit definitions are provided for the meta-information concepts and the *roles* that entities may play in a type repository community. The meta-information concepts are presented in Table 3.5, while the roles can be summarised as *users, authors* and *publishers* of types and type systems. Notably, types and type systems are immutable, once published. This is crucial to maintain the consistency of replicated meta-information elements that are disseminated in an open distributed environment. Importantly, the Type Repository Function also emphasises the need for *universal* type management [Iyengar 1998], based on a federation framework with the following variations:

- *type system interworking*: enables relationships between types across different type systems within the same repository;
- *type repository interworking*: enables relationships between types across similar type systems in different type repositories (but with the same meta-meta-model);
- *type repository federation*: enables relationships between similar types across type systems supported by distinct repositories, with separate meta-meta-models.\(^3\)

**Information specification.** This specification is given in terms of the MOF Model and the MOF abstract mapping. It describes the objects that represent type definitions and relationships, as well as the way they are expressed in MOF. In addition, it also describes how invariant and dynamic schemas are represented in the MOF.

**Computational specification.** Again, the computational specification does not introduce new features in relation to those already specified by the MOF. In particular, the following parts of the MOF constitute the computational specification:

- IDL definitions of the objects corresponding to constructs of the MOF Model;
- the Reflective package, which defines the meta-model independent interfaces;
- the MOF-to-IDL mapping; and
the MOF Repository Framework, which defines the facilities needed for the management of type repositories and their meta-models.4

Table 3.5 – Concepts defined by the ODP Type Repository (TR) Function

<table>
<thead>
<tr>
<th>ODP TR concept</th>
<th>Scope and definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR type system</td>
<td>target concepts used to define type systems; equivalent of the MOF Model</td>
</tr>
<tr>
<td>description</td>
<td>(although, in principle, other meta-meta-models can also be used)</td>
</tr>
<tr>
<td>type system</td>
<td>defines the target concepts and relationship types used in a particular domain,</td>
</tr>
<tr>
<td>description</td>
<td>according to a TR type system description; equivalent of MOF meta-models</td>
</tr>
<tr>
<td>type description</td>
<td>a typing predicate defined according to exactly one type system description;</td>
</tr>
<tr>
<td></td>
<td>equivalent of M1-level entities in the MOF</td>
</tr>
<tr>
<td>relationship</td>
<td>a predicate relating type descriptions; can be either stored in the type repository</td>
</tr>
<tr>
<td></td>
<td>or dynamically computed; equivalent of instances of M2-level MOF associations</td>
</tr>
<tr>
<td>relationship type</td>
<td>the type of a relationship, expressed in terms of the number and (meta-)types of</td>
</tr>
<tr>
<td></td>
<td>the related types; equivalent of MOF M2-level associations</td>
</tr>
<tr>
<td>relation</td>
<td>the set of all relationships of the same type</td>
</tr>
<tr>
<td>type repository</td>
<td>provides storage for: the TR type system description, type system descriptions,</td>
</tr>
<tr>
<td></td>
<td>and type descriptions, as well as for relationships and relationship types</td>
</tr>
<tr>
<td>type repository</td>
<td>a configuration of objects fulfilling well-defined roles in order to meet the</td>
</tr>
<tr>
<td>community</td>
<td>goals of type management; a type repository community can contain exactly one type</td>
</tr>
<tr>
<td></td>
<td>repository, conforming to a single TR type system description</td>
</tr>
<tr>
<td></td>
<td>(although a community can contain multiple type systems)</td>
</tr>
</tbody>
</table>

Finally, the Type Repository Function also identifies the target concepts of RM-ODP that should be modelled as meta-types. These meta-types are organised into a set of related type systems, which model different aspects of ODP systems, as defined, e.g., by the ODP computational viewpoint, the Interface References and Binding Framework [ITU-T/ISO 1998b], and the Trading function [ITU-T/ISO 1997].

3.4.4 Discussion

The three standards examined in this section represent a spectrum of approaches to meta-information management, with respect to generality and flexibility. At one extreme, the Interface Repository has a fixed meta-model, which represents the CORBA type system. The Meta-Object Facility then extends the ability to manage meta-information by using a meta-model independent approach, allowing different meta-models to co-exist in the same meta-information domain. In addition, the MOF

3 This is useful when mapping heterogeneous repository technologies, although complete mappings may not always be possible.

4 The MOF Repository Framework is currently a subject for the MOF 1.4 Revision Task Force [Crawley 2000].
specifies how to generate CORBA-compliant repositories and tools, via a standard MOF-to-IDL mapping. Finally, the RM-ODP Type Repository function provides a framework that extends the MOF approach with support for the co-existence of different meta-meta-models, by means of type repository federation. While this can be seen as an extension of the MOF approach towards a fifth layer of meta-modelling (M4), it can also be argued that the MOF’s companion standard, XMI, provides just the mechanism for realising this extra level of flexibility. XMI enables non-MOF applications and repositories to access MOF-based meta-information, by means of a mapping into the widely used XML. In addition, XMI opens the possibility for a reverse mapping, thus allowing MOF-based repositories to access meta-information defined in foreign meta-meta-models. This effectively enables type repository federation in a heterogeneous environment, without the actual need of an M4 layer. Therefore, it is reasonable to consider the two standards, the Type Repository Function and the OMG meta-modelling framework (the MOF augmented with XMI), as having the same degree of expressiveness. It is important to emphasise, however, that the Type Repository function specialises the MOF for use in ODP environments.

In the context of this thesis, these three standards play a crucial role to define the proposed approach for middleware meta-information management. While the Type Repository function provides the overall framework and identifies the needs and the policies for meta-information management in middleware, the MOF supports the definition of meta-information and the development of concrete facilities for its management. The Interface Repository in turn provides the basic structure from which to derive a complete middleware meta-model. This approach is developed in chapter 4. Note, however, that these are not the only available options on which to base a meta-information management facility (see next section). Nevertheless, the OMG and RM-ODP based frameworks have the advantage of being widely accepted standards, which is important, considering the eventual need to interchange meta-information in an open environment.
3.5 Other related technologies and standards

3.5.1 The Open Information Model and the Microsoft Repository

The Open Information Model (OIM) initiative, originally developed by Microsoft (in 1996) as part of its repository technology, and subsequently transferred (in 1998) to the Meta-Data Coalition (MDC), represents a significant industry effort towards the standardisation of meta-modelling facilities [MDC 1999]. The OIM is an extensible meta-model, based on UML, which standardises specialised sub-meta-models covering the needs of several important subject areas, such as analysis and design, object and component descriptions, databases and data warehousing, knowledge management and business engineering. The modelling process is similar, in principle, to that of the MOF. Models are specified using UML or a UML extension that corresponds to the chosen OIM sub-meta-model. Then, tools can be used for the validation of the models and for the generation of repository objects representing the models, as well as for the production of XML DTDs for model interchange. Importantly, the UML also constitutes the meta-meta-model (layer M3) for describing, interpreting and extending the OIM meta-models.5

The main differences between the OIM and the OMG MOF, apart from the different meta-meta-model, is the fact that the OIM standardises the meta-models at layer M2, while the MOF concentrates on describing, rather than standardising this layer. Note, however, that the recent merge of the Meta Data Coalition into the OMG (in September 2000) resulted in the discontinuation in the OIM initiative. This may represent the merging of the two efforts, especially regarding the sub-meta-models defined as part of OIM. In fact, the OMG CWM standard was largely derived from the OIM data warehousing sub-meta-model.

Specifically regarding the Microsoft Repository [Bernstein et al. 1999] (currently known as the Meta-Data Services that are part of Microsoft SQL Server), which is the most expressive implementation of the OIM, a few features are worth of note. First, it includes comprehensive support for the versioning of repository objects, including version history. Note that, while the MOF acknowledges the need for versioning, its current release does not specify how it can be supported. Another difference is that the

5 Although the most expressive implementation of the OIM, the Microsoft Repository, seems to use a proprietary meta-meta-model for this purpose, know as the Repository Type Model.
Microsoft Repository (and the OIM) is defined on top of concrete database support (either Microsoft SQL Server or Microsoft Access), while the MOF leaves this as an implementation-dependent issue. The main disadvantage of the Microsoft Repository (and indeed of the OIM), however, is the lack of support out of the Windows environment, as both its model and implementation rely heavily on the COM object model and infrastructure.

### 3.5.2 CDIF

The Electronic Industries Association (EIA) has proposed the Case Data Interchange Format (CDIF), as a means to standardise the exchange of modelling information used in analysis and design [Ernst 1997]. The approach is similar to the MOF, in that it explicitly uses a meta-modelling architecture with four layers, along with a standard meta-meta-model (layer M3). In fact, the MOF approach has been in part derived from CDIF. Unlike the MOF, however (and similarly to OIM), CDIF also standardises the meta-model layer (M2), which is populated by a set of standard meta-models, each targeting a different subject area. Collectively, these meta-models are known as the CDIF Integrated Meta-model. This approach, although providing standardised support for a range of application domains, tends to make the standard rather cumbersome and difficult to learn, in contrast to the cleaner approach of the OMG. This has largely contributed to the discontinuation of the CDIF standardisation process, although one cannot discard the possibility of using subject area meta-models standardised in CDIF as the basis for future OMG modelling standards.

### 3.5.3 Java Meta-data API

The Java community process is currently working on a meta-information management standard for Java, which will be known as the Java Meta-data API [Eckerson and Manes 1999; Sun 2001]. Instead of defining a new architecture, the Java Meta-data API project is based on the MOF. This is done by defining a MOF-to-Java mapping, which is similar to the MOF-to-IDL mapping, and enables the direct access and construction of MOF-based repositories in Java. The definition of the Java Meta-data API is expected to leverage the use of the MOF in the Java environment and, consequently, on the Internet as a whole. Conversely, it will also facilitate the large-scale use of Java for the management of meta-information.
3.6 Summary

This chapter is instrumental in providing the background in meta-information management for use in the approach proposed in the thesis. The overall concepts of meta-information management were presented, with emphasis on their applications and, especially, on standards and technologies for their realisation. In particular, the chapter considered the main uses of meta-information management techniques in the context of middleware, under the headings of type and configuration management. The three technologies and standards that are most influential to the thesis were then examined: the CORBA Interface Repository, the OMG Meta-Object Facility and the RM-ODP Type Repository Function. The aim was to describe their features and demonstrate their complementary nature (which will be explored in Chapter 4). For completeness, the chapter also considered other similar approaches to meta-information management, comparing them with the ones mentioned above.

The need to integrate meta-information management facilities in middleware platforms is primarily determined by requirements related to the definition, management and reuse of platform configuration descriptions. In addition, the need for semantic interoperability across heterogeneous platforms further drives the integration of these two technologies [Iyengar 1998]. Besides, there is also the need to provide services for middleware applications to define and handle application-specific meta-information in a standard way. As discussed, the three standards examined in section 3.4 provide the necessary support for these requirements.

Nevertheless, while such facilities provide an adequate foundation for middleware configuration, the fact that they work at an intensional level prevents their use for dynamic reconfiguration. As seen in the previous chapter, dynamic reconfiguration of middleware requires treatment at the level of extensions, for which object-oriented reflection is more appropriate. The goal of this thesis is to propose an approach where the two techniques are combined in order to support static configuration and dynamic reconfiguration in a unified and principled way, as can be seen in the next two chapters.
Chapter 4  The Meta-ORB Meta-Model

4.1 Introduction

The ability to flexibly configure middleware platforms, as seen in the previous chapter, can be supported with the use of meta-information management as an architectural principle. In this chapter, this principle is applied to the design of the Meta-ORB reflective middleware architecture, which is an extension of the Open-ORB architecture developed at Lancaster University [Blair et al. 1998; Blair et al. 2001]. Notably, Open-ORB is extended with an explicit meta-model, described using the MOF, which defines the kinds of entities used in the constitution of the platform, as well as the meta-information structures used to describe them. The use of such meta-information to describe and instantiate customised platform configurations qualifies the Meta-ORB as a highly configurable framework for middleware.

The role of the Meta-ORB meta-model, however, is more generic, as it is neutral with respect to the distinction between application and platform, and base- and meta-level. This means that the meta-model defines the programming model used to build application and platform configurations, considering both their base-level aspects and the associated reflection functionality. In addition, the meta-model also provides the meta-information framework for the platform. This means that it also defines the logical structure for a meta-information repository, as well as the necessary support for the management of meta-information. However, these features are defined at an abstract level, leaving their concrete definitions as implementation details (described in Chapter 6). Finally, the meta-model also identifies the concepts and aspects of the platform that are subject to reconfiguration, as will be explored in the next chapter.

The chapter is structured as follows. Section 4.2 presents the overall scope and structure of the meta-model, while section 4.3 considers the core infrastructure concepts that are required for its support. Section 4.4 presents a detailed description of the meta-model, following its modular structure and considering its support for meta-information management. Finally, section 4.5 summarises and discusses the main features of the meta-model, as well as its role in the context of the thesis.
4.2 Overall organisation of the meta-model

4.2.1 Outline

This section presents a high-level description of the Meta-ORB meta-model. It starts by delimiting the scope of the meta-model, in terms of the aspects of the platform that are represented as first-class meta-information elements. The modular structure of the meta-model is then presented, in terms of a number of MOF packages, which are organised according to the distinct platform aspects that are represented.

4.2.2 Scope

The aim of the meta-model is to provide the definition and explicit representation of the entities used to build particular configurations of the Meta-ORB platform. This is done in terms of meta-information that describes the structure of such entities, as well as the relationships among them. Importantly, the kinds of meta-information of interest coincide with the platform's self-representation, as maintained by the mechanisms for structural reflection (described in Chapter 5). Figure 4.1 classifies and illustrates the kinds of meta-information that the meta-model represents. Note that this does not represent an attempt to categorise the constructs used in middleware platforms in general. Instead, such classification is provided as a means to support the development of the approach proposed in this thesis.

Figure 4.1 – Kinds of meta-information represented in the meta-model
The elements of the meta-model correspond to the constructs of the Meta-ORB programming model (or type system), which has been influenced by CORBA and also by the RM-ODP computational language (in particular, by the interpretation of the standard as proposed in [Blair and Stefani 1997]). The first-class constructs in the meta-model are interfaces, component objects (or, simply, components) and binding objects. The corresponding meta-model elements (meta-types) represent both the type and template aspects of such constructs, meaning that the meta-model provides a basis for the functions of type and configuration management, described in Chapter 3. In this thesis, however, instances of such meta-types will simply be referred to as types or type definitions (although their definitions also include template meta-information).

The meta-model also includes elements to define auxiliary types, which do not correspond to first-class entities in the platform, but are essential to their description. Examples include: media types, constructed types and, because the type system is concrete, primitive types. In addition, the meta-model includes non-type-related meta-model elements. These elements correspond to the scope-defining constructs of the type system (e.g., module) and to auxiliary constructs, used in the definition of the first-class meta-types (e.g. operation, flow, signal and QoS annotation). Note that the RM-ODP Type Repository Function standard [ITU-T/ISO 2000] identifies a similar set of meta-types, although representing the RM-ODP computational meta-model and restricted to the type aspect.\footnote{However, in [ITU-T/ISO 2000], interface references and local bindings are represented as part of the RM-ODP meta-model. In the Meta-ORB these concepts are considered primitive services instead.}

A complete definition of the meta-model is presented, in abstract form (i.e., without committing to a particular notation), in section 4.4. On the other hand, a concrete definition is provided in terms of an Object Definition Language (ODL) for the Meta-ORB, which introduces a concrete textual notation for the definition of meta-information according to the meta-model. This language is an extension of OMG-IDL and its grammar is presented in Appendix B.

### 4.2.3 Modular structure

The structure of the meta-model is derived from the standard CORBA Interface Repository, examined in the previous chapter. The reason for this is the fact that the Meta-ORB type system is an extension of its CORBA counterpart. Consequently, the
basic meta-model elements have identical (or compatible) corresponding elements in the CORBA type system, as seen in 4.4.1. This qualifies the Meta-ORB meta-model as backward compatible with CORBA (this will be further considered in Chapter 7).

In terms of modularisation, the Meta-ORB meta-model is divided into seven interrelated packages, as illustrated in Figure 4.2 below.

![Package structure of the Meta-ORB meta-model](image)

**Figure 4.2 – Package structure of the Meta-ORB meta-model**

The BaseIDL package represents the features directly introduced from the standard CORBA meta-model. Note that primitive types are defined separately, in the BasicTypes package, imported by BaseIDL. However, as the set of primitive types is identical to that of CORBA, this package is not discussed further here. The other five packages in turn are specified in terms of the definitions in BaseIDL. These packages represent the actual extensions that characterise the Meta-ORB platform, in terms of meta-types for defining media types, QoS annotations, interfaces, components and binding objects. Notably, the dependency links between these packages illustrate the relationships between the concepts they define. For instance, component and binding definitions depend on interface definitions, which in turn depend on QoS and media definitions. Each of these packages is described in detail in section 4.4, along with their representation in UML. For brevity, however, the diagrams do not show the constraints on the specified meta-types, nor the complete list of operations they define. These features are presented in Appendix B, in a complete representation of the meta-model, presented in terms of MODL, the language used for defining MOF metamodels in the dMOF tool [DSTC 2001].
4.3 Core infrastructure

4.3.1 Rationale

In order to support the high-level constructs of the meta-model, the Meta-ORB architecture includes a minimal set of core constructs, namely, local bindings, interface references and implicit bindings. These are the primitive concepts of the programming model, and are built into the platform implementation. Thus, their definition is fixed and not subject to the configuration facilities. Note that a similar approach is adopted in Sumo-ORB [Blair and Stefani 1997]. In this section, an abstract definition of these concepts is presented, whereas their concrete implementation is discussed in Chapter 6.

4.3.2 Local bindings

A local binding is a primitive artefact used to directly connect two interfaces located in the same address space, with the requirement that communication through the local binding must be both instantaneous and reliable [Blair and Stefani 1997]. Local bindings are essential to support the connection of the interfaces of components participating in a configuration, such as seen in 4.4.5. In addition, they enable the connection of target interfaces to the respective endpoints in distributed binding objects (see 4.4.6). Importantly, in order that two interfaces can be successfully connected by a local binding, they must be strictly compatible, as defined in 4.4.2. The reason for this is that a local binding is not capable of handling incompatibilities between different interfaces. The only exception to this rule is in the case of local bindings connecting target interfaces to the endpoint interfaces of binding objects. However, in this case any discrepancies between the interfaces are handled by the internal functionality of the binding object (rather than by the local binding itself).

Thus, the infrastructure must provide a local binding service, with the following operation: \texttt{localBind(interface1, interface2)}, where the two interfaces are referred to by their interface references (see below). A result status may be returned, indicating success or failure of the operation.
Chapter 4 – The Meta-ORB Meta-Model

4.3.3 Interface references and implicit binding

According to the RM-ODP standard for interface references and binding [ITU-T/ISO 1998b], an interface reference is a structured, unambiguous and location-independent identifier for an interface. Typically, an interface reference contains or implies an interface type and contains sufficient information to allow binding to the referenced interface. It is important, however, to determine the exact contents of interface references, as this is essential for the interoperability between distinct implementations of the platform. Thus, the following information is contained as part of an interface reference in the Meta-ORB: (a) the unique name of the referenced interface, (b) the identifier of the interface type (enabling its retrieval from a repository), and (c) information about the location where the interface resides.

An interface reference is automatically created when the respective interface is instantiated. Copies of the interface reference can then be distributed to interested parties, using some external mechanism, such as a naming or a trading service, or simply as operation parameters.

Interface references are used in the Meta-ORB with the prime purpose of allowing binding factories to locate and type-check the interfaces to be bound. Interface references are also used as the arguments to the localBind operation described above, as well as to support implicit binding [ITU-T/ISO 1995b; Blair and Stefani 1997]. Regarding implicit bindings, an interface reference serves as a proxy that enables direct invocation of the referenced interface, without needing an explicit binding action. Note, however, that this is meaningful only for server operational interfaces, as discussed in [ITU-T/ISO 1995b].

4.4 Detailed description of the meta-model

4.4.1 Basic constructs

Definitions and structure

The fundamental elements of the Meta-ORB meta-model are derived straight from the standard CORBA Interface Repository, and are represented in the BaseIDL package, show in Figure 4.3. The design is influenced by the package with same name presented as part of the CORBA Component meta-model [OMG 1999b], which
considerably streamlines the structure of the IR (without affecting its functionality). However, only the features that are part of the definition of the IR as of CORBA 2.2 are included. This means that constructs for defining value objects and CORBA components are not modelled here. This is because the Meta-ORB currently does not have an objects-by-value feature, and defines its own component constructs.

![Diagram of the Meta-ORB model elements](image)

**Figure 4.3 – The BaseIDL package**

The main meta-model elements defined in this package are `Contained` and `Container`, which form the basis for the hierarchical organisation of a meta-information repository for the Meta-ORB. `Contained` is the super-class for all named objects in a repository hierarchy, defining the basic features of such objects: name, unique repository identifier, version number, absolute name, and containing
Chapter 4 – The Meta-ORB Meta-Model

repository. Container in turn is the super-class for all objects that can contain other objects in the hierarchy, and defines the basic features for navigating and searching for objects through the hierarchy. Note that containers are also Contained objects, except for the root container in the hierarchy, which is represented by the Repository meta-model element. This is possible by the introduction of an auxiliary meta-model element, TopContainer, which defines the features common to all containers. In addition, the module also defines IRObjec, which is the base class for all objects in the repository (called repository objects), including the anonymous ones.

Besides the organisational constructs, the BaseIDL package defines the basic meta-types of the Meta-ORB type system, notably for defining constructed types, such as struct, union, array, sequence and enumeration. (The complete set of primitive types of CORBA 2.2 is then imported from the BasicTypes package.) The BaseIDL package also includes the abstract meta-type IDLType, from which all type definitions are derived. This meta-type represents the kinds of types that can be used to specify attributes, operation parameters, return values and signal values, as seen below. In addition, a further abstract meta-type is defined: Typed, which is the base class for all meta-model elements that have an associated type.

Importantly, note that the meta-model elements related to interface definitions are not represented in this package. They are instead part of the Interfaces package, as the concept of interface is re-defined in the Meta-ORB.

Support for meta-information management

The meta-model elements defined in the BaseIDL package provide the basic support for managing meta-information, including functionality to create, name and access meta-information elements.

Access to meta-information is provided through the basic lookup and navigation facilities defined by the meta-model elements Container, Contained and Repository. From a given repository object (derived from Contained), it is possible to navigate to its container or to the root repository object. From a container, in turn, it is possible to search for any repository object, using the operations described in Table 4.1, which are identical to those defined for the CORBA IR.
Table 4.1 – Standard repository access operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup_name</td>
<td>Recursively searches for a repository object in the hierarchy, based on its non-scoped name (see below) and on other parameters that specify the number of levels to search, the kind of object being searched, and if inherited objects are to be considered or not (in case the container object is an interface definition).</td>
</tr>
<tr>
<td>lookup</td>
<td>Searches for a repository object based on its scoped name (see below).</td>
</tr>
<tr>
<td>lookup_id</td>
<td>Defined only for the root container (i.e., instance of Repository), enables the search for a repository object based on its unique repository identifier (see below)</td>
</tr>
<tr>
<td>contents</td>
<td>Enumerates the objects defined in the scope of a container</td>
</tr>
</tbody>
</table>

Note, however, that alternative operations for repository access also exist, as a result of applying the MOF-to-IDL mapping. In addition, particular implementations of the repository can also define their own access operations, in order to suit particular needs.

Regarding the creation of meta-information, meta-model element TopContainer (as well as other kinds of containers) includes operations to create each of the kinds of repository objects defined in the meta-model, using the style defined for the CORBA IR. For instance, operation create_op_interface enables the creation of a repository object to represent an operational interface type. The complete set of create operations is presented in Appendix B. In addition, as with access operations, alternative operations to create repository objects exist, as defined by the MOF-to-IDL mapping.

Importantly, as in [ITU-T/ISO 2000], the repository has to ensure that meta-information elements (notably types), once published, are immutable. This is mainly because types represent contracts about their instances, and their integrity is essential to support dynamic type checking in a reliable way. Furthermore, there is no causal connection between a type and its instances, meaning that any changes to the type would result in its representation of its instances becoming inconsistent, which would undermine dynamic type checking and type management in general.

Finally, with respect to the naming of repository objects, which is a prerequisite for type management (see Chapter 3), the elements of the BaseIDL package use three different kinds of names (the exact format of these names is specified in Chapter 6):

- *non-scoped names*, which uniquely name repository objects within the scope of their respective containers;
Chapter 4 – The Meta-ORB Meta-Model

- *scoped names*, which consist of concatenated non-scoped names that uniquely identify repository objects relatively to a given point in the repository hierarchy (such as the root, in case of *absolute scoped names*, or an intermediate container);

- *repository identifiers*, which are globally unique identifiers that enable individual repository objects to be unambiguously identified across independent repositories.

4.4.2 Interfaces

Overview

The Interfaces package, shown in Figure 4.4, represents the constructs used to define the interfaces of components and binding objects in the Meta-ORB. An interface definition represents a contract between a component and its users, defining the services it offers, their properties and the way they can be accessed. An instance of an interface definition then represents an access point onto the respective component.

Three styles of interfaces are supported, as in RM-ODP: *operational*, *stream* and *signal* interfaces. A base class for interface definitions is provided (InterfaceDef), in order to allow interface definitions to be treated in a generic way\(^2\). This base class defines the features that are common to all interface styles:

- *interface inheritance* – interface types can be defined by inheriting from one or more other interface types, following the same rules as defined in the CORBA type system, and as long as the base and derived types are of the same style; and

- *inheritance-based sub-typing* – similarly to its equivalent in the CORBA IR, the \texttt{is\_a} operation returns true if the interface type on which it is invoked is either identical or inherits from the interface type identified as the argument\(^3\).

In addition, the interactions defined in interface definitions can have associated QoS annotations (defined in section 4.4.4). This is introduced via the abstract meta-model element \texttt{QoSConstrained}, from which the meta-model elements representing interactions are derived. In addition, interface definitions can have attributes, which are defined by \texttt{AttributeDef}, in an identical way as in standard OMG IDL interfaces.

\(^2\) Note that this is not the same as InterfaceDef in the CORBA IR.

\(^3\) The test is based on the repository identifiers of the two interface types.
Chapter 4 – The Meta-ORB Meta-Model

Operational interfaces

This interface style is represented by the OpInterfaceDef meta-type, and is used for request-based interactions. Similar to interfaces in CORBA 2.2, operational interface definitions can contain operation, attribute, constant, typedef and exception definitions. Operations are defined using the auxiliary meta-model element OperationDef, which is similar to operation definitions in CORBA, but with two extensions. First, an operation definition has an explicit causality attribute, which can
be either provided or required (in CORBA, only the former is implicitly supported). Second, an operation can have an associated QoS annotation, which specifies targets for the performance of interactions using the operation (e.g., round-trip delay and error rate). The other features of operations correspond to the standard ones defined in CORBA 2.2: mode (either normal or oneway), result type (through Typed), parameter list, exceptions and contexts (represented as strings).

In addition to the above features (and to those inherited from InterfaceDef), OpInterfaceDef has an associated role attribute. This is a derived attribute, which represents the generic role that interfaces play in interactions: client (with only required operations), server (with only provided operations), or client-server (with both kinds of operations). OpInterfaceDef also defines an operation (compatible) to type-check interfaces for structural compatibility, in order to determine if two interfaces can be directly bound to each other by a local binding (discussed in section 4.2). Structural compatibility of operational interfaces is determined by the following rules, which define the complementarity of operational interface types:

- every operation in one interface must have a matching counterpart in the other interface, and vice-versa (note that dangling operations are not allowed); and
- the signatures of matching operations must be identical, except for their operation causalities, which must be opposite.

Note that the above rules cannot be relaxed (e.g., with the introduction of sub-typing), as local bindings are not able to perform type and QoS conversions. Interface sub-types are however allowed in explicit bindings, as seen in section 4.4.6 (although the mechanisms used for type checking are distinct from the one described above).

Stream Interfaces

This interface style, represented by the StrInterfaceDef meta-type, is aimed at supporting continuous media interactions, thus qualifying the Meta-ORB as a multimedia middleware framework. Similarly to operational interfaces, stream interface definitions can contain attribute, constant, and typedef definitions (but no exceptions). However, the kind of interactions contained in stream interfaces consists of flows, rather than operations. Flows are specified using the FlowDef meta-type, and represent the unit of continuous media interaction (an abstraction over a sequence of
more primitive interactions, usually constrained by a timing relation). A flow has a
media type, such as audio or video, which is specified through the FlowType
association with a MediaSpecificationDef element (defined in section 4.4.3). In
addition, flows are unidirectional (from a producer interface into one or more
consumer interfaces), as represented by the direction attribute (valued either in or out
for a given interface). Similarly to operations, flow definitions can also be constrained
by QoS annotations, which define bounds for the performance of flow interactions,
such as throughput, delay and jitter.

As with operational interfaces, stream interface definitions also have a compatible
operation, to determine whether two stream interfaces are structurally compatible for
local binding, as defined by the following rules:

• each flow in one interface type must have a matching flow in the other interface
type, and vice-versa (note that dangling flows are not allowed in local bindings);
• the signatures of matching flows must be identical, except for their directions,
which must be opposite.

Signal Interfaces

This interface style is represented by the SigInterfaceDef meta-type, which
supports the most primitive kind of interactions in the Meta-ORB. Signal interfaces
may contain attribute, constant, and typedef definitions (but no exceptions), as well as
signal definitions. A signal is an atomic interaction, typically used for real-time
communication between a signal producer and one or more consumers. Signals are
defined using the SignalDef meta-type, which has two attributes, direction and values.
The direction attribute determines whether the signal is emitted (out) or received (in)
in the interface on which it is declared. The values attribute in turn specifies the data
values, if any, carried by an occurrence of the signal. Interestingly, signals can be used
as primitives in terms of which to define the more complex interaction styles:

• oneway operations – a pair of signals: request emission, by the client interface,
and request reception, by the server interface;
• normal operations – a sequence of four signals exchanged between client and
server: request emission, request reception, reply emission and reply reception;
Chapter 4 – The Meta-ORB Meta-Model

- flows – a continuous sequence of emission-reception signal pairs, one for each packet of data that is sent as part of a flow.

Finally, SigInterfaceDef also defines a compatible operation, to enable the type checking of local bindings between pairs of signal interfaces, as defined below:

- each signal defined in one interface type must have a matching signal in the other interface type, and vice-versa (dangling signals are not allowed); and
- the signatures of matching signals must be identical, except for their directions, which must be opposite.

4.4.3 Media types

Although this is not a major focus for this thesis, the Meta-ORB meta-model defines a flexible scheme for media type definitions, in order to support multimedia interactions. The scheme is based on the concepts of major and minor media types (as in the Microsoft DirectShow toolkit [Microsoft 2000d]). The meta-types for media definition are represented in the MediaDefs package, show in Figure 4.5.

Major media types represent the broad categories of media data supported by the platform. They are specified in the MajorMediaTypes enumeration, and currently, the following ones are defined: audio, video, image and animation. Major media types,
however, only consist of simple identifiers. The actual media type properties are instead specified in the minor media types, which are defined in a two-step process.

First, *generic (minor) media types* specify the kinds of media *encoding* that are supported in a given instance of the platform, along with their property names, types and range of acceptable property values. These generic types are used to type-check as well as to help generating the definitions of specific (minor) media types (described below). Definitions of generic media types are kept in a *MediaTypeSystem* container.

The next step is then to define the *specific media types* that are ultimately used to qualify multimedia interactions. They are defined as instances of the meta-type *SpecificMediaTypeDef*, which is a specialisation of the concept of generic media type. A specific media type contains the same properties (attributes) as the generic type it specialises, but the values for these properties can be refined from the values specified in the definition of the generic type. In addition, the meta-model also defines the concept of a *media specification*, which consists of a container of specific media types, and works as the actual type given to flows, signal values, operation parameters and return values. A media specification provides a set of alternative specific media types for an interaction, where the actual type used will depend on a media type negotiation (which can be part of the process of binding, described in section 4.4.6).

### 4.4.4 QoS annotations

Quality of Service (QoS) has been generically defined, in the context of RM-ODP, as a set of quality (or non-functional) requirements on the collective behaviour of one or more objects [ITU-T/ISO 1996]. In addition to this generic definition, a more precise model for QoS definition and management has been the subject of further RM-ODP related standardisation, resulting in the OSI/ODP Quality of Service Framework [ITU-T/ISO 1995c] and the QoS in ODP standard [ITU-T/ISO 1998a]. In particular, the Quality of Service Framework defines the basic terminology for QoS definition, which includes the following concepts:

- **QoS categories** – logical classification of QoS requirements; the most important QoS categories from the point of view of multimedia middleware are timeliness, volume and reliability [Blair and Stefani 1997], although other categories are also defined, such as security, safety, coherence and cost;
Chapter 4 – The Meta-ORB Meta-Model

- **QoS characteristic** – an independent and quantifiable aspect of QoS, within a given category, such as delay and delay variation (for timeliness), throughput (for volume), and frame and bit error rates (for reliability); and

- **QoS attribute** – an attribute of an object expressing its QoS with respect to a particular QoS characteristic.

In addition, the QoS Framework also defines the concept of QoS management, as the set of activities (performed by QoS mechanisms) to support the monitoring, control and administration of QoS in a system.

Although QoS is not a major issue in this thesis, the Meta-ORB meta-model defines a simplified scheme (derived from the QoS Framework concepts described above) for the specification of QoS meta-information associated with flows, signals and operations. Such meta-information can then be used by eventual QoS management mechanisms implemented on top of the Meta-ORB platform. The meta-types for QoS definition are represented in the QoSDefs package, shown in Figure 4.6 below.

![Figure 4.6 – QoSDefs package](image)

The QoSDefs package does not specify a particular set of QoS characteristics (or even categories). Instead, the generic constructs used to specify such features are defined. QoS characteristics are specified using the QoSCharacteristic meta-model element. A QoS characteristic is defined in terms of its name, the kind of values it can receive (target value, range value to maximise, range value to minimise, maximum
acceptable value, minimum acceptable value), and the respective unit of measurement (e.g., milliseconds, megabits per second, percentage, absolute value). Note that a data type is not specified, as QoS attributes conforming to a QoS characteristic can only receive numerical values (double-precision floating point) in the current version of the Meta-ORB. In addition, note that the interpretation of QoS characteristic descriptions is left entirely to the QoS management mechanisms, although a degree of standardisation is generally assumed. Importantly, QoS characteristics are grouped in a singleton instance of the QoSBaseDef meta-model element, which is the registry of the existing QoS characteristics, used to validate QoS attribute definitions.

Finally, the actual QoS attributes that are associated with flows, operations and signals are defined as instances of the meta-model element QoSAttribute. However, QoS attributes can only be defined as part of a QoS annotation, which groups together the set of QoS attributes to be associated with a given interaction definition. A QoS attribute is defined in terms its name and associated values, which must conform to an existing QoS characteristic declared with the same name in the QoSBaseDef object.

### 4.4.5 Components

**Overview**

Components are the unit for the encapsulation of state and functionality in the Meta-ORB programming model. A component exposes its functionality in terms of one or more well-defined interfaces, which are the only points at which the component can be accessed. However, unlike typical component models [Szyperski 1998], where the concept of a component exists only as a deployment unit, in the Meta-ORB components exist both at design time (as deployment units) and at runtime (as uniquely identifiable entities). As will be seen in the next chapter, the existence of components as runtime entities is crucial to enable dynamic adaptation of platform configurations. A runtime component is the product of the instantiation of a component type, performed with the use of a component factory service. The graphical representation for a component and its interfaces is illustrated in Figure 4.7.
Chapter 4 – The Meta-ORB Meta-Model

Figure 4.7 – Graphical representation for components and their interfaces

As described below, two kinds of components are supported, *primitive* and *composite* components. The meta-modelling elements supporting their definition are shown in Figure 4.8 below.

**Primitive Components**

Primitive components are the basic building blocks in the Meta-ORB. A primitive component is an atomic encapsulation of a language-specific implementation, providing language-independent interfaces for access to its functionality.
A primitive component type is defined by means of the `PrimComponentDef` metatype, whose attributes specify the language-specific implementation of the component, in terms of its name (e.g., the name of the programming language class representing the implementation) and the load-able implementation code (which can be either a copy of the code itself or a reference, such as a URL, to a file where the implementation code can be found). In addition, a primitive component type also specifies the types and names of the interfaces of its instances, through the related `PrimInterface` meta-model element. Note that the same primitive component type can have interfaces of different styles (e.g., a stream interface for multimedia production, and an operational interface for control).

**Composite Components**

This kind of component represents the means for composing functionality in the Meta-ORB. A composite component is an encapsulated *configuration* of internal components, providing a single external view, through one or more interfaces, for accessing the functionality of the configuration as a whole. Thus, from the outside, composite and primitive components are indistinguishable, although their internal definitions are dramatically different. Importantly, a composite component can comprise an arbitrarily deep composition hierarchy, where some of its internal components can themselves be composite. In addition, the interfaces of a composite component correspond to a *mapping* (or exposure) of selected interfaces of its internal components. An example composite component, with its internal configuration, is shown in Figure 4.9.

![Figure 4.9 – A composite component and its internal configuration](image-url)
Composite component types are defined by means of the CompComponentDef meta-type, which has two one-to-many aggregation relationships. The InternalComps relationship associates a composite component definition with the meta-model element InternalComponent, used to specify the name and the type of each of the internal components. Note that the types of the internal components are independently defined, and can be either primitive or composite component types. The ComplInterfs relationship in turn relates the composite component type with the meta-model element ComplInterface, used to specify the name, the type (through the inherited ComplInterfType relationship) and the mapping of each of the component’s interfaces. The mapping of the composite component’s interfaces is in turn defined in terms of the name of the internal component exposing the interface, as well as the name of the interface in the context of this internal component.

Finally, the meta-type for composite components has an attribute to specify the internal configuration, represented as an object graph, similarly to the approach introduced in [Hokimoto et al. 1996]. The graph is specified as pairs with the form:

\[(\text{comp}_\text{name1}, \text{interf}_\text{name1}) : (\text{comp}_\text{name2}, \text{interf}_\text{name2})\]

meaning that the named components are connected through their nominated interfaces. In addition, the following constraints must hold on a component configuration:

- an interface of a component can only be connected to a single other interface, and this connection is made through a local binding (see section 4.2); and
- any interface that is mapped as an external interface of the composite component cannot be bound to other interfaces inside the configuration.

Component definition and instantiation

The meta-model (and the corresponding repository) offers the basic support for component configuration. It does so by providing an abstract syntax and semantics for component definition, as well as meta- and meta-meta-information to enable the verification of consistency of component types (by the type definition tools) and the creation of component instances. For composite components, consistency checks are carried out by (a) retrieving the types of all internal components (along with the types of their interfaces) and verifying the validity of any local bindings between them, and
(b) checking if the configuration rules defined in the component meta-type are respected. For primitive components, in turn, consistency is verified by type-checking the component implementation (at the level of the programming language used), in order to ensure that it correctly supports the interfaces of the component.

Instantiation in turn is performed by a *component factory*. In particular, the creation of a *composite component* and its interfaces is performed in a recursive way, as the factory is re-invoked to create each of the internal components. Thus, the factory must also have access to meta-information describing the types of the internal components. After creating each of the internal components at a given level of composition, the factory performs the local bindings between their interfaces (according to the object graph), and maps their appropriate interfaces as interfaces of the composite. Finally, the factory returns (to its caller) a unique *identifier*, which can later be used to refer to the created component. The creation of primitive components, on the other hand, is implementation-dependent and is described in Chapter 6.

### 4.4.6 Bindings

**Overview**

The Meta-ORB meta-model supports the concept of *explicit binding*, which enables the connection of components, through their interfaces and independently of their location. Explicit binding is characterised by the need to carry out a binding action before interaction is enabled [ITU-T/ISO 1995b]. The result of this action is the creation of a *binding object*, which supports interaction between the bound interfaces according to a contractual context [ITU-T/ISO 1995b; ITU-T/ISO 1996]. Importantly, the existence of such an explicit context for interactions (i.e., the binding object) is an essential feature of multimedia middleware platforms, as it enables the management of the QoS associated with multimedia interactions [Blair and Stefani 1997].

The precise specification of a framework for bindings has been the subject of further standardisation in the context of RM-ODP [ITU-T/ISO 1998b]. The meta-model for bindings in the Meta-ORB has been influenced by this framework, especially regarding the entities participating in binding establishment, as well as the information model (i.e., meta-model) for bindings and the binding establishment process itself. A binding establishment action involves the following kinds of entities:
• *binding initiator* – the entity requesting the creation of the binding, which can be either one of the components to be connected (first-party binding) or an independent component (third party binding);\(^4\)

• *binding factory* – which performs the binding action, upon request from the binding initiator;

• two or more *target interfaces* – the interfaces to be bound, which are identified through their interface references (see section 4.2); and

• the binding object resulting from the action.

Finally, the coordination of these entities is done according to a *binding protocol*, which defines the sequence of actions involved in binding establishment (see below).

**Binding objects**

Bindings are distributed objects, typically spanning two or more address spaces, with the main purpose of connecting the interfaces of interacting components. A binding object encapsulates mechanisms realising all the aspects involved in component interaction, such as:

• basic communication, by conveying the individual elements (packets) of interactions from producer interfaces into consumer interfaces, typically using some standard transport protocol;

• multiplexing and de-multiplexing of interactions, which may involve processing of the streams of data conveyed by the binding (e.g., inter-media synchronisation);

• transformations on the data packets, as in the case of filtering (e.g., compression and decompression, encryption and decryption, etc); and, more generally,

• provision of the non-functional properties of interactions, such as QoS.

Binding objects are thus the essential elements in the provision of middleware services. However, from the above description, it can be concluded that different applications will have different requirements for binding, implying that no single binding structure will satisfy all kinds of applications. Thus, the Meta-ORB does not prescribe a fixed binding structure (as in traditional middleware platforms). Instead,

\(^4\) The binding initiator is assumed to be the owner and controller of the binding.
its meta-model provides the constructs used in the definition of custom binding configurations, as shown in Figure 4.10, which presents the Bindings package.

![Diagram showing the Bindings package](image)

**Figure 4.10 – The Bindings package**

As shown in the diagram, two kinds of bindings are supported: *primitive bindings* and *composite bindings*. The distinction between these two kinds is similar to the distinction between primitive and composite components, and is described below. Despite this, the two kinds of bindings have a few features in common, illustrated in Figure 4.11 below, which shows the graphic notation for a binding scenario.
As the figure suggests, a binding object can have multiple endpoints\(^5\), each one supporting the connection of a different component, possibly in a different address space. The endpoints of a binding are specified as binding role definitions, which can be seen as descriptions of “sockets” where target interfaces of a conforming type can be plugged. A given binding type is defined in terms of one or more role definitions, each one defining a template for binding endpoints. A particular instance of a binding type may then have zero or more endpoints conforming to a given role definition, depending on the set of target interfaces to be bound and their types. Thus, a binding type defines an abstract configuration, which can generate different binding objects, each with a different concrete configuration. Note that similar approaches to the definition of binding types, based on role definitions, are adopted in [Lindsey and Linington 1996], [Williams and Arnold 1997], and [Berry and Kaplan 1998]. However, these related approaches only consider the overall structure and semantics of binding roles, rather than also including their precise internal configurations, as in the Meta-ORB meta-model.

The definition of a binding role is given in terms of the expected target interface type and the configuration for the role (although these two features depend on the kind of binding, as seen below), along with the following features (see Figure 4.10):

- **matching_rule** – the actual types of interfaces to be bound must match the expected target interface type associated with the role definition. This attribute represents the rule used to determine such matching. The following matching rules are currently supported:
  - **STRICT**: the type of the target interface must be an exact mirror of the expected target interface type (this is the default matching rule);

\(^5\) Although binding endpoints are simply interfaces onto the binding object, the different graphic notation illustrates the fact that they are defined in a different way.
– **SUBTYPE**: the expected target interface type must be a sub-type of the actual type of the target interface (i.e., the binding role must support all the features provided and required by the target interface), where sub-typing is based on inheritance, as defined by the is_a operation of InterfaceDef (note that a similar approach to sub-typing for bindings is adopted in [Berry and Raymond 1994], although it also considers the behaviour of the involved interface types); and

– **PARTIAL**: some of the interactions declared in the expected target interface type need not necessarily be supported by the actual target interface; the list of optional interactions is declared in the optional_interactions attribute.

- **cardinality** – this attribute constrains the number of times the role definition can be instantiated in a binding (i.e., the number of endpoints that conform to the role). Values may be given either as an exact number (e.g., 1) or as a range (e.g., 1..5).

- **causal_depend** – in some cases, the realisation of a role definition at an endpoint requires or excludes the realisation of another role elsewhere in the binding (e.g., a consumer role requires the existence of a producer role). This attribute specifies such dependencies by naming the roles on which the current role depends and by specifying the nature of the dependency (*required* or *excluded*).

**Primitive bindings**

Primitive bindings correspond to the fundamental level of a binding configuration and typically have the purpose of encapsulating transport protocols, enabling their services to be used via the higher-level and protocol-independent abstraction of interfaces. Similarly to primitive components, primitive bindings are atomic entities and, although they can be distributed across several address spaces, they have no internal structure and their implementations are opaque. This is because primitive bindings have language- and system-dependent implementations, which are outside the scope of the middleware platform, as far as their internal details are concerned.

The specification of a primitive binding type is done in terms of its implementation code (given either in the form of the actual load-able code or as a URL from where the actual code can be downloaded). A second attribute then gives the name of the programming language entity (e.g., a class) that represents the entry point to the implementation. When creating a primitive binding, its implementation code is loaded
and started in each of the binding endpoints. In addition, as a container, a primitive binding type can contain *primitive role* definitions (PrimRoleDef), which define the types of endpoints that instances of the primitive binding type can have. In addition to the attributes that are common to both kinds of roles (as seen above), a primitive binding role includes a reference to the type of target interfaces expected by the role.

Importantly, primitive bindings cannot be used in isolation in the Meta-ORB (although, in principle, this would be feasible). Instead, their use is restricted to the internal structure of more complex bindings (as described below). In addition, as can be seen from the above description, primitive bindings do not have a control interface, as is the case with composite bindings.

**Composite bindings**

Composite bindings, also called *open bindings* in [Fitzpatrick et al. 1998], are binding objects with an explicit internal structure, in terms of a configuration of components and nested bindings. As with composite components, the internal configuration of a composite binding is amenable to inspection and adaptation, and is specified in terms of an *object graph*, although the nodes can be other binding objects, as well as component objects. Importantly, as a composite binding can have other composite binding objects nested in its configuration, it follows that this kind of binding can be structured in multiple levels. This is useful when the interaction mechanisms need to be organised according to levels of abstraction, where the innermost level only provides an interface to the raw transport protocol, while the levels above provide services that are progressively more sophisticated. An example is shown in Figure 4.12, which presents a hypothetical three-level composite binding.

![Figure 4.12 – A multi-level composite binding](image-url)
Chapter 4 – The Meta-ORB Meta-Model

The meta-type for composite bindings (CompBindingDef) is defined as a container, which can only contain composite binding role definitions (defined by the meta-model element CompRoleDef). In addition, a composite binding definition can reference one or more internal binding declarations (defined by the meta-model element InternalBinding), each one specifying the name and the type of a nested binding (note that only immediately nested bindings are declared).

The specification of a composite binding configuration is done in a distributed way, where each of the role definitions contained in the binding type is defined with a partial configuration. When instantiating the binding type, the partial configurations of the realised roles (for each of the binding endpoints to be created) are combined, resulting in the complete configuration of the binding. The definition of composite binding roles is done as follows. The basic attributes are inherited from the base class RoleDef (described above). In addition, the following features are defined:

- **InternalComps** – this aggregate relationship enables the declaration of the names and types of the components contained in the configuration of the role;
- **configuration** – in a similar way as for composite components, this attribute specifies the configuration (object graph) of a composite binding role, in terms of a list of pairs \([(obj\_name, interface)\), \((obj\_name, interface)\)], although, crucially, at least one of the named objects in the list must refer to one of the nested bindings declared for the binding type where the role definition is contained; and
- **RoleTargetInterf** – this aggregate relationship enables the declaration of the expected target interface type that endpoints conforming to this role definition can bind, as well as the mapping of the endpoint’s interface (as an interface of an internal component of the role configuration).

Finally, a composite binding can also have one or more control interfaces, which enable an external entity (e.g., binding owner) to interact with it as if it were a component. The utility of the control interfaces, as the name implies, is to provide control over the binding operation, such as with operations to start and stop the binding. The existence of multiple control interfaces allows the control functionality to be partitioned among independent interfaces, each with a different purpose.
Chapter 4 – The Meta-ORB Meta-Model

Binding definition and instantiation

Static management of binding configurations is performed using the same principles as for components. The structural consistency of a binding configuration is checked during the creation of a binding type (by a type definition tool, using the appropriate meta-type). A binding configuration is then effectively created by a binding factory, when instantiating a binding type. However, because a binding configuration is expressed in an abstract way (which must be actualised depending on the types and number of the target interfaces to be bound), the instantiation of a binding type involves extra considerations. Furthermore, as bindings are distributed objects, the process of creating their configurations must be performed through a collaboration of entities (e.g., binding factory fragments) located at the several endpoints of the binding and co-ordinated by a binding protocol.

A binding protocol defines the sequence of actions performed by the entities taking part in the binding establishment process, especially the binding factory, which executes the major part of the protocol. The Meta-ORB, however, does not prescribe a fixed binding protocol (in the same way as it does not prescribe a rigid binding configuration). Instead, implementations of the platform are free to define these features in the most suitable way, as long as they respect the definitions and rules of the meta-model. For instance, different binding protocols may adopt different strategies for determining the properties of a binding (e.g., QoS negotiation) or for performance optimisation. Therefore, the Meta-ORB architecture defines an abstract binding protocol, derived from [ITU-T/ISO 1998b] and [Blair and Stefani 1997], which serves as a framework for defining custom binding protocols. This protocol is illustrated in Figure 4.13, followed by a description of the steps it involves.

Figure 4.13 – Outline of the abstract binding protocol
1. The binding initiator requests the creation of the binding to the binding factory. The initiator passes, as arguments, the identifiers of the target interfaces to be bound (e.g., their interface references), as well as an optional binding type identifier.

2. The binding factory locates the target interfaces and performs the necessary type checking, which may involve the choice of a suitable binding type (if not given by the initiator), as well as the selection of appropriate roles for the endpoints corresponding to each of the target interfaces. This step may also involve the negotiation of binding properties (such as QoS) among the target interfaces, as well as the negotiation of media types (if multimedia interactions are involved).

3. The factory creates the binding configuration, by instantiating the required components and nested bindings at each of the binding endpoints. This step is executed recursively, by re-applying the protocol to create the nested bindings.

4. The target interfaces are locally bound to the interfaces of the respective endpoints.

5. A reference to the control interface of the created binding is returned to the binding initiator (if the binding has more than one control interface, the first one is returned, which is considered the main control interface).

Finally, in case of a failure in any of these steps, an error is reported to the initiator and the partial binding may be destroyed, depending on the severity of the error (i.e., in case the error compromises the whole binding).

### 4.5 Summary

This chapter has presented the programming model of the Meta-ORB platform, in the form of a MOF-based meta-model. The design is driven by the principles of *flexibility* and *modularity*. This means that the meta-model defines a framework for developing customised platform configurations, in terms of well-defined and reusable building blocks, notably, components and binding objects. In addition, the meta-model also includes configuration rules, which provide for the structural consistency of platform definitions. These features characterise the Meta-ORB as a highly configurable middleware architecture, justifying its name as an abstract infrastructure (a meta-platform) for defining concrete platform instances. Fundamentally, the meta-
model applies seamlessly to the design of both middleware and application configurations. While this may blur the distinction between application and platform functionality, it crucially contributes towards the openness of the architecture.

Other important features of the meta-model refer to its flexible support of quality of service and media type definitions, which qualifies the Meta-ORB as a multimedia middleware architecture. In addition, the programming model is language- and system-independent, permitting its realisation in different environments, while retaining the same expressiveness. Notably, the constructs of the meta-model enable the encapsulation of application and platform implementations in the form of language-neutral units, contributing towards interoperability and reuse. Furthermore, the meta-model is backward compatible with the CORBA meta-model, implying that standard IDL 2.2 definitions can be used in the Meta-ORB.

In the overall context of the thesis, the definition of the meta-model serves three complementary purposes (besides being a representation of the platform's programming model). Firstly, it is the foundation for the meta-information management facilities of the Meta-ORB. In fact, the basic support for type and configuration management is defined as part of the elements of the meta-model. Secondly, the meta-model provides the constructs (components and interfaces) used to define the reflective architecture of the Meta-ORB. Finally, the meta-information represented by the meta-model is used as a basis for the self-representation of a platform, as maintained by the mechanisms for structural reflection. These aspects will be revisited in Chapter 5, which describes the meta-level architecture, and in Chapter 6, which presents a concrete implementation of the several aspects of the platform.
Chapter 5  Meta-ORB: Reflection Framework

5.1 Introduction

The meta-information framework and associated meta-model presented in the previous chapter define the essential support for static platform configuration in the Meta-ORB. Customised platform configurations can be defined in terms of meta-information elements that represent the entities comprising the platform. Dynamic re-configuration, on the other hand, requires some means to manipulate such meta-information at runtime, in a way that is causally connected with the specific instances of platform configuration. This is the role of the reflective meta-level, which completes the architecture of the Meta-ORB and contributes towards fulfilling the requirements identified in Chapter 2.

In this chapter, the design of the meta-level is described from a conceptual point of view, without committing to a particular implementation. The fundamental properties and the structure of the meta-level are discussed in section 5.2, with emphasis on the use of a multi-model reflection framework (MMRF), first introduced in the context of middleware by the Open-ORB architecture [Blair et al. 1998]. Section 5.3 presents the foundation for integrating the reflection and the meta-information management frameworks. Importantly, this integration forms the basis for a unified treatment of configurability and dynamic reconfigurability in the Meta-ORB architecture. This is explored further in section 5.4 in terms of the relationship between reflection and type evolution, which is one of the central ideas of the thesis. Section 5.5 then discusses related work specific to this area, while section 5.6 concludes with a discussion of the main features and highlights of the approach.

5.2 Structuring the meta-level

5.2.1 Basic principles

Reflection in the Meta-ORB is used for dynamic inspection and adaptation in the context of both platform and application elements. To this end, the overall architecture
is conceptually divided into base-level, where the actual functionality of the platform and applications is defined, and meta-level, where the reflection capabilities are defined. The design of the meta-level follows the principles of the Open-ORB reflective middleware architecture [Blair et al. 2001], which are discussed below.

The most fundamental principle of the meta-level is referred to as object-oriented reflection, which means that the entities that populate the meta-level are uniquely identifiable objects. The definition of object, in the Meta-ORB, refers to the structural constructs defined in the meta-model, namely components, binding objects and interfaces, although, typically, only component objects (and their interfaces) are used in the constitution of the meta-level. Hence, the meta-object protocol (MOP) is realised in terms of the interfaces of components that play the role of meta-objects. In addition, object-oriented reflection also assumes that the base-level is similarly structured in terms of objects, meaning that meta-objects are used to reify components, binding objects and interfaces. Interestingly, this can be seen as a form of meta-circularity, where the same programming model applies to both base- and meta-levels.

The second property of the meta-level is an implication of object-oriented reflection: meta-objects are associated with objects on a per-object basis. This means that each base-level object can have its private meta-object or set of meta-objects, which are collectively referred to as the object’s meta-space. The Meta-ORB is thus referred to as having a reflective architecture with per-object meta-spaces. Among other consequences, this means that the scope of reflective computation performed by a given meta-object is limited to the base-level object to which it refers. This tends to be the appropriate approach in reflective middleware (see discussion on the scope of reflection in Chapter 2). In addition, this approach is also important when considering the issue of the safety of reflective computation, as the effects of reflection can be limited to individual base-level objects. In this way, the risks of breaking the consistency of the whole platform as a result of misleading reflective computations are significantly reduced. Note, however, that the granularity of base-level objects may vary greatly. At one extreme, meta-objects can be assigned to a whole composite binding configuration. At the other extreme, they can be assigned to fine-grained primitive components representing only a small element of the platform or application. An illustration of this discussion is provided by Figure 5.1 below.
Crucially, meta-objects are created on demand (i.e., upon the first time a particular meta-object needs to be accessed). This implies that not all base-level objects need to have associated meta-objects, and that the overhead of reflection is only incurred if the corresponding reflective functionality is needed.

The next property of the meta-level refers to the possibility of stacking several levels of “meta”, in what is known as a reflective tower. This is possible because, as seen above, meta-objects are built using the same constructs as for the base-level, which means that they can have (meta-)meta-objects of themselves. Thus, it is possible to use the MOP to inspect and adapt the meta-level itself.

Finally, a procedural reflection style is adopted, which essentially means that reflective computation is used to manipulate the actual implementation of a platform or application. This gives freedom to the meta-level programmer to introduce completely new behaviour in the platform implementation (i.e., not limited to options specified a priori). However, note that constraints may be defined, which help to ensure the consistency of reflective computations (by not allowing them to proceed if they are to produce inconsistent results). This topic will be revisited in section 5.4.

5.2.2 Multi-model reflection framework

Definitions

The adoption of a multi-model reflection framework (MMRF) is one of the most distinctive characteristics of the Open-ORB architecture [Blair et al. 2001], being also inherited by the Meta-ORB. The fundamental principle of multi-model reflection was introduced by [Okamura et al. 1992] in the AL-1/D language (see Chapter 2) and
consists in applying *separation of concerns* to the design of the meta-level itself. Following this principle, the meta-space of an object in the Meta-ORB is partitioned into a number of distinct aspects, each one realised by a different meta-object. This is important to reduce the complexity of the meta-level, especially considering the multitude of features that must be managed in a meta-level for middleware.

Each separate aspect of the meta-level is defined in terms of a *meta-space model*\(^1\), which represents the structure and functionality for the reification of a base-level object according to that aspect. In the Meta-ORB architecture, however, the definition of a given meta-space model is given at an abstract level, although with a clear meaning and scope of application. The actual details, notably the exact meta-object protocol associated with each meta-space model, are thus an issue to be solved in particular implementations of the architecture, as will be illustrated by the prototype described in Chapter 6.

Figure 5.2 below illustrates the concept of using distinct meta-objects (each one corresponding to a different meta-space model) to reify a given base-level object.

\[ \text{Figure 5.2 – Reifying a base-level object according to multiple meta-space models} \]

**Consistency of the meta-space**

In principle, the different meta-space models are independent from each other. This means that the meta-level programmer can choose the most appropriate one for a particular situation and then work on it in a way that is transparent from the aspects represented by other meta-space models. As demonstrated by [Okamura 1995], this

---

\(^1\) Note the distinction from the similar term "meta-model", used for meta-information management.
greatly simplifies the task of programming at the meta-level, especially considering concurrent and distributed systems.

However, such independence between the meta-space models is only completely true from an external point of view (i.e., from the perspective of enabling the isolation of concerns when programming at the meta-level). From the perspective of the effects on a base-level object, interference may happen between the meta-space models which may affect the correctness of reflective computation. This is due to concurrency between the meta-objects corresponding to the different meta-space models and to dependencies between them, especially when they manipulate shared elements of the base-level. Nevertheless, as also demonstrated by [Okamura 1995], preconditions can be defined as part of a MMRF, in order to constrain the concurrent execution of meta-objects so that the consistency of base-level objects is ensured. The issue of dependencies between meta-space models will be revisited in section 5.2.4.

**Partitioning the meta-space**

Significantly, the above discussion stresses the importance of carefully designing the meta-level, in order to avoid unnecessary dependencies between its meta-space models. In addition, if carefully identified, dependencies can be managed in order to ensure the consistency of the base-level. While a rigorous treatment of such dependencies is outside the scope of this thesis, this general principle for the partition of the meta-space is adopted.

In particular, five meta-space models are specified, with well-defined abstract design and semantics, which enable the relevant dependencies to be clearly recognised. The meta-space models are categorised according to the usual distinction between *behavioural* and *structural* reflection, discussed in Chapter 2. The behavioural part of the meta-space consists of two meta-space models: *Resources* and *Interception*. The former is responsible for reifying and managing the resources (such as storage and processing) used by base-level objects, while the latter deals with the manipulation of implicit behaviour associated with the interfaces of components (e.g., pre- and post-processing that affect the non-functional properties of the interactions on a given interface). Structural reflection, on the other hand, is represented by three distinct meta-space models: *Interface Discovery* (which reifies the set of interfaces supported by a component or binding object), *Interface* (which reifies the constitution
of a particular interface), and Architecture (which reifies the internal configuration of a component or binding object).

Collectively, these meta-space models enable the representation of the most relevant aspects of a middleware platform, at least from the perspective of supporting applications with dynamic requirements and environments. This thesis, however, concentrates only on the structural meta-space models, which are at the centre of the relationship between reflection and meta-information management (presented in section 5.3). The structural meta-space models are described in detail in 5.2.3 below.

The behavioural meta-space models, on the other hand, are the subject of related work conducted at Lancaster. In particular, the design of the Resources meta-space model is described in [Duran-Limon and Blair 2000a], while the Interception meta-space model was explored in the first prototype of the Open ORB architecture [Costa et al. 2000a].

Importantly, however, this is not a fixed set of meta-space models, implying that the meta-level architecture can be extended with new meta-space models, e.g., in order to introduce different aspects for reification or to reify existing aspects from a different perspective.

5.2.3 Structural meta-space models

Interface Discovery meta-space model

This meta-space model enables access to the external view of components and binding objects. In the case of components, this external view corresponds to the interfaces that a component provides, whereas in the case of binding objects, it is related to the endpoints and the control interfaces of a binding. The kind of access provided by this meta-space model is restricted to introspection (i.e., modifications to the base-level object are not allowed), with the purpose of enabling the discovery of the features exposed by a component or binding object. This is mainly because changes to the external view of an object usually require the manipulation of the object’s internal structure, which is outside the scope of this meta-space model (instead, this is a concern for the Architecture meta-space model, as seen below).

Note that in the original specification of the meta-space structure (in the overall Open ORB architecture) these two meta-space models (Interface Discovery and Interface) are combined into a single one, which is referred to as the Interface meta-space model.
For component objects, the functionality provided by meta-objects of this meta-space model is straightforward, typically consisting on the ability to enumerate and search the interfaces of a component, based, e.g., on their identifiers and types. This is similar in purpose to the generic \texttt{IUnknown} interface for discovering the application-specific interfaces of an object in COM [Microsoft 2000a].

For binding objects, on the other hand, besides the above functionality applied to the binding control interfaces, the \texttt{Interface Discovery} meta-space model also enables access to the endpoints of a binding. This includes facilities to enumerate the binding endpoints, along with their respective properties such as the endpoint location, the expected target interface type and the rules for matching this type with that of the actual bound interface. This kind of functionality is useful, for example, to support the evolution of an application when some of its components, linked by a common binding object, need to be replaced (e.g., to determine compatibility of a new application components with its corresponding binding endpoint).

**Interface meta-space model**

This meta-space model provides access to the structure of individual interfaces of components and binding objects. Again, only \textit{introspection} is provided, as interfaces in the Meta-ORB represent stable contracts between interacting parties and, therefore, are immutable.

The purpose of this meta-space model is similar to the Java Core Reflection API [Sun Microsystems 2000]. Its functionality consists of convenient operations to access the definition (i.e., the type) of an interface, without the need to directly interact with the meta-information repository. This allows the discovery of the features of an interface, such as attributes and interaction definitions (i.e., operations, flows or signals, depending on the style of the interface). Examples of such operations include enumeration and search of attributes and interactions, as well as inspection of their descriptions (cf. signatures). In addition, the \texttt{Interface} meta-space model also supports \textit{direct access} to attributes (to get and set their values) and operations (for dynamic invocation). This enables a given interface to be accessed without static knowledge of its definition. Note that this represents a dynamic form of implicit binding, and is the equivalent of the dynamic invocation interface (DII) of CORBA.
Architecture meta-space model

This meta-space model, in contrast to the other two, enables access to the internal structure of components and binding objects. It is especially important in the context of this thesis as it enables both introspection and adaptation, therefore offering support to the evolution of platform and applications. Importantly, only composite components and bindings can be reified according to this model. This is because the internal implementation of primitive components and bindings is a language- and system-dependent issue, which is, by definition, out of scope in middleware.

Reification according to this meta-space model is mainly achieved in the form of the object graph that represents the internal configuration of the base-level object.\(^3\) In addition, as defined in [Blair et al. 2001], architectural constraints (which help maintaining the consistency of a configuration as it evolves over time) and explicit architectural styles may also be part of the meta-level representation. This means that this meta-space model effectively enables the manipulation of the software architecture of a component or binding object. In the Meta-ORB, however, the focus is on the object graph only, and architectural constraints and styles are not explicitly represented. Instead, the fundamental constraints related to the composition style defined by the meta-model are typically hard-coded in an implementation of the Architecture meta-space model. Work on a complete meta-representation of software architectures is again the subject of related research being conducted at Lancaster [Moreira et al. 2001].

The functionality of the Architecture meta-space model thus consists of operations to inspect the structure of a configuration, as well as to modify it. Inspection essentially enables the identification of the components that are part of a configuration, as well as the interconnections (local bindings) between them. Modifications to the configuration can then be made by the insertion, replacement or removal of components, as well as by manipulating the local bindings connecting them. For components, such configuration adaptations have obvious meaning. For binding objects, however, they assume two different forms: endpoint-based adaptation, where the modifications are isolated to a single binding endpoint, and role-based adaptation, where all the existing endpoints that conform to the changed role are

---

\(^3\) The use of object graphs here has the same meaning as defined in Chapter 4, i.e., a set of objects (components or binding objects) interconnected by their interfaces.
modified at once. Importantly, the (implicit) constraints specified in the Meta-ORB meta-model (such as type compatibility) must not be violated by these operations.

In addition, the Architecture meta-space model enables new interfaces to be added to components and binding objects. This is typically done by adding and connecting the internal component(s) needed to support the interface, and then mapping the appropriate interface of the new internal component as an external interface of the composite. Note, however, that existing interfaces cannot be removed, so that the adapted component is guaranteed to support the services that were originally provided.

5.2.4 Additional considerations

Access to the meta-space

Access to the meta-space of a given base-level object is provided in a separate way for each of the meta-space models. Typically, the infrastructure provides a set of operations (one for each meta-space model), which, when called, return a reference to the interface of the respective meta-object. These operations, which are collectively referred to as the basic MOP, require an argument that identifies the base-level object to be reified, as seen below for the structural meta-space models:

- *Interface meta-space model* – access is based on the identifier of the reified interface, which is part of its interface reference; and

- *Interface Discovery and Architecture meta-space models* – access is based on the identifier of the component or binding object being reified.

Identifiers are therefore used as tokens when granting access to the meta-space. This allows a minimal level of security, by ensuring that only entities that know the proper identifier can use the facilities of a given meta-space model to access a particular base-level object. Such access control to the meta-space is crucial, especially in the case of the Architecture meta-space model, as modifications to the reified object should be restricted to authorised parties. In addition, by using two kinds of tokens, two levels of access control are provided. For the Interface meta-space model, access is considered public, as an interface reference can be acquired through public means (e.g., through a name server or trader). For components and bindings, however, access is more restricted since their identifiers are not publicly available. In principle, the identifier of a component or binding object is only known to the owner
of the object (i.e., the entity that invoked the component or binding factory for its creation), as well as to other parties to which the identifier is explicitly granted.

Note that this is an over-optimistic approach to security, and a more complete treatment to the issue (e.g., through more sophisticated access control and authentication mechanisms) would be necessary. Such treatment, however, is outside the scope of this thesis.

Dependencies between the meta-space models

In the case of the structural meta-space models described above, dependencies are minimised, as only the Architecture meta-objects can perform changes to the base-level. Currently, a single dependence is identified:

“When a new interface is added to a component or binding object, using the Architecture meta-space model, the view offered by the Interface Discovery meta-space model is affected.”

This means that the Architecture meta-object must immediately notify its Interface Discovery counterpart so that it can update its representation of the base-level object.

However, if considering a complete meta-level picture, including the behavioural meta-space models, further dependencies must be treated. In particular, the Resources meta-space model may be affected when changes are made via the Architecture meta-space model, such as when components with allocated resources are replaced or removed. In addition, adaptations made through the Interception meta-space model may also affect its Resources counterpart, as new or changed interceptors may alter the consumption of resources. In the context of this thesis, though, such dependencies will not be further considered.

Sources of adaptation

The meta-space models, in particular the Architecture model, offer the necessary support for structural adaptation. However, the decision about the way adaptations should be carried out, as well as the conditions that trigger them, are again outside the scope of the meta-space models. The need for adaptations is typically triggered by dynamic changes in the platform environment (such as variations in network bandwidth), as well as in the user requirements. In both cases, reaction to such
changes (in terms of suitable adaptations) can be controlled either by some automatic mechanism or by direct user intervention.

In the former approach, a monitoring and control mechanism (such as the one used in [Blair et al. 2000a], which is based on timed automata machines) can be used to enable the timely detection of predefined adaptation conditions and the appropriate reaction in terms of calls to the meta-object protocol. This is the typical case when the supported application has real-time needs, such as the quality of service requirements of multimedia applications. In the latter approach, interactive tools are usually provided in order to enable the meta-level user to directly invoke the facilities of the meta-object protocol. A typical situation for this happens when changes in user requirements demand intelligent decisions about what kinds of adaptations should be performed in response. In addition, it is likely that this approach is usually applied when the results of adaptation are of a more permanent nature, while the automatic approach is better suited to transient adaptations. Importantly, in both approaches the mechanisms to control adaptation should be defined in terms of the Meta-ORB metamodel, so that they can themselves be subject to reflection and adaptation, if required.

5.3 Reflection and meta-information management: principles for an integrated approach

5.3.1 Outline

This section explores the fundamentals for the integration between reflection and meta-information management, discussed in generic terms in Chapter 3. The relationship between the extensional style of meta-objects and the intensional style of type definitions is examined, with emphasis on the meta-information that is common to both of them. In particular, types are seen as defining and providing the self-representation maintained by meta-objects, which in turn provide the facilities for this meta-information to be manipulated in a way that is causally connected with the represented base-level objects. In addition, this section also looks at another aspect of the integration, namely, the definition of meta-objects using the meta-information management facilities and the representation of the link between base- and meta-level.
5.3.2 The reflection and the meta-information hierarchies

The use of meta-information management results in a hierarchy of types and metatypes, which corresponds to the layered structure of the meta-modelling architecture. In a similar way, a parallel hierarchy emerges from the use of object-oriented reflection, consisting of meta-objects, meta-meta-objects, and so on. Crucially, as shown in Figure 5.3, the two hierarchies have a common base, which consists of the objects represented by types and meta-objects. In addition, the two hierarchies are orthogonal to each other, as they represent two essentially distinct forms of reflection, respectively intensional and extensional (see Chapter 2). While a type represents a group of objects (i.e., all its instances), a meta-object typically represents a single object. Thus, distinct instances of the same type, as shown in the figure, can be independently represented and manipulated by their own individual meta-objects.

![Figure 5.3 – The two hierarchies of meta](image)

This relationship is exploited in the Meta-ORB, by defining the self-representation of base-level objects, as maintained by meta-objects, to be closely related to the respective type definitions. The consequences of this are explored in sub-section 5.3.3 below, which considers the reification process. However, despite representing essentially the same meta-information, types and meta-objects use it in different ways. From the perspective of a type, as seen in Chapter 4, this meta-information is mainly used for object instantiation and type checking. Thus, in the context of a type, meta-information is read-only. Meta-objects, on the other hand, use the same meta-information for dynamic inspection and, in the case of the Architecture meta-space model, for adaptation of the particular base-level objects they reify. Hence, for a meta-object, such meta-information is mutable, and changes to it happen in a causally
connected way, with the corresponding effects on the base-level object. The implications of maintaining these two uses of meta-information simultaneously are investigated in section 5.4, where the issues of type evolution are considered.

Note, however, that the above-described relationship between the two hierarchies is conceptually restricted to the level of types (i.e., below the dashed line in Figure 5.3). This means that there is no direct relation between meta-types and meta-meta-objects. Instead, all meta-object levels are considered in the same way, with the self-representation they maintain being derived from the types of their respective base-level objects. For instance, the self-representation maintained by a meta-meta-object is related to the component type defining the reified meta-object. This will become clearer when discussing meta-object definition in sub-section 5.3.4.

5.3.3 Initialising the state of meta-objects

Basic principle

As implied by the above discussion, the type of a base-level object contains, either directly or indirectly, the necessary meta-information to allow reification. This is especially true considering the structural meta-space models, which are the focus in the thesis. Consequently, this meta-information can be used to initialise the state of meta-objects. The details of the initialisation process, however, depend on the kind of base-level object being reified, as will be seen next. Nevertheless, the overall process of meta-object creation is essentially the same in all cases, and involves the following steps:

1. **Instantiation.** A component representing the meta-object is instantiated by a component factory. The identifier of the base-level object and the repository identifier of its type, are passed to the component as an instantiation parameter;

2. **Type retrieval.** The meta-object queries the meta-information repository for the required meta-information about the type of its base-level object (using the repository identifier mentioned above); and

3. **Initialisation.** The meta-object processes the obtained meta-information, in order to produce an appropriate meta-level representation, which it then uses to initialise itself.
Meta-level representation of interfaces

This is the most straightforward case, partly because individual interfaces can only be reified through the Interface meta-space model. The meta-representation maintained by meta-objects of this meta-space model coincides exactly with the contents of interface types. This means that step 3 above basically consists of installing the appropriate meta-information. Furthermore, the immutability of interfaces in the Meta-ORB means that such meta-representation is also immutable.

Meta-level representation of components

Components can be reified by meta-objects corresponding to the Interface Discovery and Architecture meta-space models. In each case, the initialisation process relies on the definition of the component type plus additional runtime meta-information, as seen below.

- **Reifying the Interface Discovery meta-space** – the component type provides the names and types of the interfaces supported by the component, although their unique identifiers must be obtained from the component’s runtime representation (described in Chapter 6). Note, however, that, in the case of composite components, the set of supported interfaces may change due to reflection performed by the Architecture meta-space. This means that a mechanism must be in place to maintain the meta-representation up to date.

- **Reifying the Architecture meta-space** – in this case, the entire contents of the (composite) component type are used. In particular, the component type provides: the internal configuration of the component (in terms of its internal components and the graph showing how they are connected), the interfaces that the component supports, and the mapping of these interfaces into interfaces of internal components. Typically, the same kinds of structures used to represent these features in the component type are also used in the meta-object. In addition, as in the above case, the unique identifiers of the interfaces and internal components are obtained from the component’s runtime representation.

Meta-level representation of binding objects

Similarly to components, binding objects can be reified according to both Interface Discovery and Architecture meta-space models. In addition, meta-information is
required from both the binding type and the runtime representation of the reified binding object (see Chapter 6).

- **Reifying the Interface Discovery meta-space** – the names and types of the control interfaces of a binding, as well as the role definitions for its endpoints are provided by the binding type. On the other hand, information about each of the binding endpoints (e.g., their identifiers, roles and location) depends on the particular instance of the binding type. Therefore, this kind of information must be obtained from the runtime entity that represents the binding. In addition, the unique identifiers of the control interfaces must also be obtained from the runtime.

- **Reifying the Architecture meta-space** – in this case, the binding type represents an abstract definition of the binding configuration. It must then be combined with runtime meta-information about the binding endpoints (especially their roles and location) in order to generate a concrete representation of the binding object. More precisely, the partial configurations corresponding to the roles realised in the several binding endpoints must be joined, through the nested bindings that they have in common. In this way, a complete object graph (containing components and nested bindings) is generated, resulting in a concrete representation of the binding configuration. The process is illustrated in Figure 5.4, which shows a binding with four endpoints, two of them realising the same role. The dashed lines show how the partial endpoint configurations are joined, based on the nested bindings they have in common (named A and B). Interestingly, this process of generating the complete object graph mirrors the work of the binding factory when creating the actual binding\(^4\). This includes the process used to determine the identifiers of the internal components of the binding, with the meta-object using the same naming scheme employed by the binding factory (see Chapter 6).

\(^4\) Although the Architecture meta-object only deals with one level of the binding, while the factory deals with all levels in a recursive way
Caching type meta-information

As an enhancement to the initialisation process, a meta-object caches the type definition of its base-level object locally. This may be done either physically, by maintaining an explicit copy of the repository object corresponding to the type, or logically, by deriving such data when necessary. In the latter case, the copy of the repository object is derived from the meta-representation held by the meta-object, via the reverse of the process used during initialisation. In this way, subsequent accesses to the repository can be avoided, as the cached copy can be used for general purposes, when access to the type is required. Importantly, since meta-information published in the repository is immutable, there are no problems of cache coherence in this case.

In addition, the maintenance of a cached copy of the type is crucial to the process of type evolution, as described in section 5.4.

5.3.4 Meta-object definition

As already mentioned, meta-objects are specified in terms of component types, as defined by the Meta-ORB meta-model. Thus, there are typically a number of component types, which correspond to realisations of the several meta-space models. This implies that new meta-object types can be introduced, for instance, in order to provide different MOPs that suit specific user needs when accessing a given meta-space model. As another implication, the fact that meta-objects are themselves components means that they can also be reified using other meta-objects. Figure 5.5 illustrates this approach to meta-object definition by means of a generic example.
Figure 5.5 – Defining the meta-level objects in terms of the Meta-ORB meta-model

The above figure also shows how the reflection hierarchy (i.e., the tower of meta-objects) fits in the context of the layered hierarchy of the meta-modelling architecture, thus complementing the view presented in 5.3.2. Interestingly, the definition of meta-objects in terms of elements of the model layer (M1) justifies the use of the term “model” when referring to each of the meta-space partitions (meta-space models).

However, the representation of the meta-space models is not complete in the Meta-ORB meta-model. More precisely, the Base-Meta relationship (between an object and its meta-object) cannot be explicitly represented, as this would require associations between modelling entities in different layers (e.g., the type of a meta-object would need to refer to the meta-type defining the kind of base-level object it is meant to reify). This is not supported by the MOF, due to its strict meta-modelling approach (discussed in Chapter 3). Hence, Base-Meta is represented in an ad hoc way, such as by using variables defined in the implementation of the meta-object’s type to hold references to the base-level object, as well as to its type.

Note, though, that an additional study conducted by the author suggests the use of a loose meta-modelling approach as a principled way to represent the Base-Meta relationship. Such an approach could be derived from the ideas proposed in [Geisler et al. 1998] and [Bezivin and Lemesle 2000] for solving the similar problem of
representing the `InstanceOf` relationship. This, however, will not be further explored in this thesis, as it would involve the adoption of another meta-modelling architecture, instead of the MOF. Another possible approach would be to define a base type for all types of base-level objects, so that this type could be referenced by the meta-object’s type. While this would be feasible for interface types, it would not be possible for component and binding types, as the meta-model does not support inheritance in these cases. Hence, this approach has also been dismissed. Nevertheless, it is important to note the potential benefits of explicitly modelling the `Base-Meta` relationship. In particular, this would provide a principled means to differentiate between base- and meta-objects, allowing the runtime mechanisms to correctly validate and enforce their capabilities and constraints (e.g., meta-objects can have access to the internals of components, while base-level access is strictly constrained by component encapsulation).

### 5.4 Reflection and type evolution

#### 5.4.1 Overall approach

As discussed in Chapter 4, type definitions are immutable once they are published in the repository, so that the consistency between a type and its instances is maintained. However, some kinds of reflective adaptation (on individual instances) result in structural changes that contradict the type of the adapted instances. In other words, the properties expressed in the type may no longer hold for an adapted instance. A solution to overcome this inconsistency is to allow an individually adapted instance to have its original type replaced by an *evolved version*, independently from other instances of the same type. In this way, by treating the new type version as a first-class type, the fundamental consistency requirement between types and their instances is preserved. Notice, however, that not all kinds of adaptation will necessarily result in such type changes, as seen in 5.4.2.

Figure 5.6 illustrates the basic dynamics of the approach. On the left-hand side, a hypothetical base-level object is shown, along with its original type and its respective meta-object. The meta-object, as discussed in section 5.3.3, happens to *cache* the definition of the object’s type. Subsequently, a request for adaptation is issued by a client of the meta-object (e.g., by the base-level object itself or by a third party). As a
result, the meta-object changes the base-level object, with a corresponding change of its type. However, only the cached copy of the type is modified (as the type definition published in the repository is immutable), meaning that the copy becomes dissociated from the original type in the repository. The result is presented on the right-hand side of Figure 5.6, which also shows the redirection of the `InstanceOf` link in order to point to the new type. This means that the principle of consistency between an instance and its type is preserved, as the new type correctly represents the adapted instance. In addition, the use of rules to prevent adaptations that would result in the creation of invalid new types, as discussed in 5.4.4, can further reinforce the consistency of the result.

![Figure 5.6 – Impact of adaptation on the type of an object](image)

Crucially, types evolved in this way have the same properties as types defined in the usual way (and stored in the repository), except for two differences. Firstly, an evolved type cannot be instantiated, meaning that its sole instance is the base-level object whose adaptation resulted in its creation. Secondly, an evolved type can be mutated as a result of further adaptations of its instance (this is not a problem in this case, as the type has no further instances). This also implies that the new version cannot be used to generate further versions of the type. However, these differences are eliminated if the new type is published in the type repository, from where it can later be retrieved and used by factories to create new instances. The issue of type publishing is considered in sub-section 5.4.3.
5.4.2 The impact of adaptation on type evolution

Foreword

The exact details of the approach described above depend on the kind of base-level object in question. As adaptation in the Meta-ORB is currently only considered in relation to the Architecture meta-space model, the effects of reflection on type evolution are limited to the types of composite components and binding objects. As seen next, such effects depend on whether the external view of the base-level object is affected or not, and whether the results of adaptation need to be reused. This resembles the discussion on change and type evolution presented in [Kilov et al. 1997], which suggests the classification of changes according to their importance in a given context, so that only important changes result in the creation of a new type.

Impact of component adaptation

As seen in section 5.2.3, two aspects of a component are subject to adaptation: the internal configuration and the set of supported interfaces. However, only adaptations concerning the set of supported interfaces (by adding new interfaces) necessarily cause the evolution of the component’s type. This is because the addition of a new interface has an impact on type management issues related to the component (such as in dynamic type checking).

In contrast, adaptations on the internal configuration of a component (such as by adding a new component) do not necessarily result in type evolution. This is because the internal configuration is encapsulated by the component, and changes to it do not affect the component’s external view. Nevertheless, the internal configuration resulting from an adaptation or sequence of adaptations may need to be reused afterwards, in order to generate new components that incorporate the results of these adaptations. In this case, the decision to evolve the type must come from the user of the meta-object, who can explicitly request the creation of a new type, based on the modified configuration. The new type is subsequently published (see below) in the repository so that it can later be reused.
Impact of binding adaptation

In the case of binding objects, three kinds of adaptations can be made through Architecture meta-objects (as seen in section 5.2.3): endpoint-based adaptation, role-based adaptation, and the addition of new control interfaces. Only the second and third kinds (inevitably) cause type evolution, while the first kind does not have such an effect (since it represents changes that are local to a given endpoint and do not propagate at the type definition level). In the case of adding new control interfaces, the same effect is observed as when adding new interfaces to components (see above).

Role-based adaptation, on the other hand, consists of changes that affect the configuration of all existing binding endpoints that conform to the changed role. Thus, evolving the role definition (and thus the binding type) will preserve the consistency between the type and the actual endpoint configurations. However, the major reason for evolving the type definition, as a result of role-based adaptation, is related to the dynamic creation of new binding endpoints (in multi-point bindings). In this case, the mechanism for creating new endpoints can rely on the current (i.e., evolved) type of the binding in order to get an up-to-date definition of the roles corresponding to the endpoints to be created.

5.4.3 Support for type versioning

Managing version numbers

When type evolution is required, instead of changing the original type definition of the adapted object, a new type definition is created. More precisely, the new type is created as a version of the original one, implying that some version management scheme must be in place. Notably, a type version is logically represented as a distinct repository object, although sharing the same absolute scoped name as the original type definition. Each different version of a type definition, however, has its own unique repository identifier, which is typically differentiated by a version number. For instance, in the repository identifier format adopted in the Meta-ORB prototype (see Chapter 6), the main part of the identifier is composed of the absolute scoped name of the object, followed by a version number.

Thus, the version management mechanism should be in charge of allocating unique version numbers to successive versions of a type. Version numbers can be defined
following a scheme of major and minor numbers, in order to distinguish slight type changes from more profound ones. In Chapter 6, a simplified scheme for the management of version numbers will be presented which offers the basic support required for type evolution and versioning. In addition, although out of scope in this thesis, a more sophisticated version model could be employed, such as the ones discussed in [Conradi and Westfechtel 1998] for software configuration management, or the scheme used in the Microsoft Repository [Bernstein et al. 1999]. This would offer greater control over the successive versions of a type, especially considering relationships between them, such as version history and differences between versions.

Publishing new versions

As seen above, a newly created version of a type is logically represented as a repository object, meaning that it can be used as a normal type. However, the new version is neither actually stored in the repository (it is only held by the meta-object that performed its creation), nor can it be used to create further instances. Nevertheless, a version may evolve to such a state when it represents a stable new configuration, which may have further utility. For instance, runtime adaptations can be made in order to make the configuration of platform and application components suitable to a new environment or to new user requirements. In such cases, the new version can be published in the repository, so that it can later be retrieved and reused in other similar situations.

Publication of a type version involves placing it, as a repository object, in the meta-information repository hierarchy. In particular, the new version is placed inside the same container where the original type definition is located. This is necessary in order to maintain the consistency of the version’s scoped name, which corresponds to its location in the repository. In addition, it facilitates location of all existing (published) versions of a given type. Importantly, after publication, a type version can be used (by factories) to instantiate new objects. This implies that the version cannot be further mutated and that any eventual adaptations (of any of its instances) have to be done through the usual process of type evolution (i.e., by creating a new, distinct type version). Therefore, for all practical purposes, a type version that has been published in this way behaves just like a normally defined type.
5.4.4 Type evolution constraints

An enhancement to the process of type evolution via reflection is the ability to define type evolution constraints, which determine the validity of new type versions. By enforcing such constraints, it is possible to avoid adaptations that would result in erroneous types (according to some criteria defined in the constraints). Type evolution constraints thus have effective control over the process of reflective adaptation, although only in cases where the use of reflection results in type evolution (see above). Importantly, such constraints can be defined in a way that their enforcement results in a given relationship to be preserved between a type definition and its succeeding versions. For instance, a type evolution constraint can introduce a sub-typing relationship, so that a type description can evolve into a new version if and only if the latter represents a sub-type of the former. Furthermore, different constraints can define different kinds of sub-typing relationships, such as structural and behavioural sub-typing. Other kinds of constraints on type evolution (which do not necessarily define type relationships) include configuration consistency rules (e.g., to enforce a particular pattern of composition) and application-specific rules (such as dependencies between particular kinds of components in a configuration). Note that this approach has similarities to the work presented in [Moreira et al. 2001] on architectural constraints, discussed in section 5.2.3.

The scope of type evolution constraints may vary from constraints defined for particular type descriptions through to constraints defined for the entire contents of a given container or even for the whole repository. When multiple levels of constraints exist, the one at the innermost level (e.g., per-type constraints) is given precedence. In addition, constraints can be defined and associated in one of three ways: declaratively (by selecting the constraint from a set of pre-defined ones and declaring it as part of meta-information definitions), procedurally (e.g., by providing consistency-check methods) or statically (by hard-coding the constraint in the implementation of the repository or even in the implementation of the meta-objects). Only statically defined constraints are currently supported, as the other two ways of defining constraints would require changes in the meta-model (such as to introduce a constraint specification clause as part of the syntax for type definitions). Furthermore, an agreement on the available kinds of type evolution constraints needs to be made, so that different implementations of the Meta-ORB can interpret them in a consistent
way (although this may not be a problem with procedurally-defined constraints, which may be portable from one repository into another).

5.4.5 Major features and benefits of the approach

The approach presented in this section exploits the fact that, although both techniques deal with meta-information that is characteristic of layer M1, *meta-information management* treats such meta-information as related to all instances of a type, whereas in *object-oriented reflection* it is specific to each individual instance. In addition, the *causal connection* link that exists between a reflective meta-object and its base-level object (which represents a one-to-one relationship) does not exist in meta-information management. This is reflected by the fact that types in the repository are immutable, and also by the fact that there is no link from a type to its instances (the only existing link in this case, `InstanceOf`, is from an instance pointing to its type). Importantly, this means that the approach is *scalable*, as there is no need to track the instances of a type in the distributed system.

The use of meta-objects therefore enables the manipulation of meta-information on a per-instance basis, allowing each individual instance to be dynamically reconfigured, independently from other instances of the same type. Nevertheless, the relationship between the meta-objects and the type repository means that essentially the same meta-information is used as the basis for static configuration and dynamic reconfiguration. This is important to guarantee the consistency between meta-information used during design (or configuration) time and runtime. The approach thus represents a contribution towards the integration of the different phases of a platform (or application) lifecycle.

To summarise, the main benefits of the combined approach in the context of the Meta-ORB architecture are:

- the use of the type repository as a logical central point for keeping meta-information, thus providing a natural facility for meta-object initialisation;
- the ability to maintain consistency between the meta-objects that reify base-level objects of the same type, since they use the same meta-information (despite the fact that these meta-objects may have different implementations or correspond to different meta-space models);
- the possibility to include type evolution constraints in the type definitions, so that
  reflective reconfiguration does not produce or allow unexpected results; and
- the possibility to dynamically define new types (by reflectively changing existing
  types) and, especially, to make them available for reuse through the repository.

### 5.5 Related work

A number of researchers have addressed similar issues to the ones considered in
this chapter, notably type evolution in open distributed systems (particularly in
relation to service types) and the relationship between reflection and type systems. In
what follows, relevant efforts in these areas are briefly described and compared to the
approach presented above.

The relationship between reflection and type systems (cf. meta-models) has also
been identified by [Stemple et al. 1993]. This work is also based on the observation
that the meta-representation manipulated by structural reflection mechanisms (which
they call *linguistic reflection*) is closely related to type-based meta-information. In
particular, type meta-information (through type checking) is used to validate the
outcome of reflective operations, in order to avoid type-incorrect changes to a system.
This is similar to the approach proposed in this thesis, although their approach is to
type-check reflectively generated program fragments, in order to ensure correctness
(instead of verifying type relationships between successive versions of a type). In
addition, their approach differs from the approach presented here in one essential
feature: it is based only on *intensional* reflection. This means that reflection is carried
out directly onto type descriptions, with effect on all instances of the modified type
[Kirby et al. 1996]. This is a result of their focus on the support for evolution in data-
intensive applications, such as database systems and persistent data stores. In contrast,
in the approach proposed in this thesis adaptations are only possible at the extensional
level (i.e., on individual instances) which, as discussed before, is more appropriate in
the context of dynamic middleware adaptation.

In [Erradi et al. 1992], an approach is presented which also combines reflection and
type evolution. Types can be dynamically modified with the use of structural
reflection facilities, which are constrained by invariant conditions specified as part of
the type system (i.e., in the meta-type). Each meta-type can define its own set of
invariants, which are enforced during type evolution, in order to guarantee the
consistency of modified types. This approach, however, differs from the one presented in this thesis in two major ways. Firstly, *published types are mutable*, as types are considered like objects that can evolve in the same way as their instances. Secondly, modifications of a type have effect on all of its instances, meaning that only *intensional* reflection is provided.

The issues of type evolution and the relationship between old and evolved types through versioning have been extensively explored in [Senivongse 1997], in the context of ODP services. This work introduces the concept of *evolution transparency*, which masks the occurrence of evolution from the clients of the evolved services. This is mainly achieved through behavioural compatibility [ITU-T/ISO 1996], by requiring the new version of a service to support (possibly via some structural or semantic conversion of invocations) clients that only know the old versions. This is similar to the use of type evolution constraints (e.g., sub-typing) in this thesis, where, for instance, an adapted component supports all the services (interfaces) of its previous type (although in this case conversion of interactions is not necessary, as only the component type is changed, not the types of its interfaces). The approach, however, is focused on the evolution of types, rather than on the evolution of particular instances, as in this thesis. In particular, although different versions of the same type can co-exist, all its instances should ideally be converted into the latest version (though this is not a mandatory requirement). Although this may be appropriate for service type evolution (to keep all servers of a given service type mutually consistent), it is not suitable for the kind of adaptations targeted in this thesis, which mainly result from the reaction to conditions that are specific to particular instances of a type (such as when supporting dynamic applications).

5.6 Summary

This chapter presented two of the central contributions of the thesis. Firstly, it described the use of a reflective architecture in the context of middleware, providing for the dynamic adaptability of the structure of platform and applications. In particular, the concept of multi-model reflection framework is exploited, which helps managing the complexity of the meta-level. Secondly, and most important, an approach was proposed to integrate the techniques of meta-information management (based on the meta-model presented in Chapter 4) and object-oriented reflection, in
the context of middleware. The different facets of this approach were described, notably:

- the use of meta-information management techniques to define the structure of meta-objects (in terms of component types), as well as their state (in terms of meta-information corresponding to the types of their base-level objects), and

- the use of reflection to dynamically define and publish new meta-information via type evolution (and as consequence of the adaptation of individual instances), thus enabling the reuse of configurations that result from reflective adaptation.

While other researchers have pointed out the relationship between reflection and meta-information management (in the form of type management), the use of reflection in these cases is intension-based (e.g., based on meta-classes). To the author’s knowledge, the combination of intensional and extensional (i.e., per-object meta-spaces) reflection, especially in the context of middleware, is a unique feature of the approach proposed in this thesis. While a purely intensional style is clearly useful in contexts such as database systems, an approach that is also capable of working at the level of extensions is more appropriate for middleware configurations, where different platform components are typically independent from other components of the same type. This is especially true when considering adaptations in dynamic environments.

Finally, it is important to emphasise that the approach presented in this chapter represents an abstract framework, which can be realised in a number of different ways. An actual implementation of this framework will be presented in Chapter 6, where concrete realisations of the structural meta-space models and meta-information management facilities are provided, along with the infrastructure to support them.
Chapter 6 The Meta-ORB: Concrete Design and Implementation

6.1 Introduction

The design of the Meta-ORB was presented in the previous two chapters, in terms of its meta-model and associated meta-information framework (Chapter 4), as well as its reflection framework (Chapter 5). The design was defined in an abstract way, enabling multiple distinct implementations. In addition, as discussed in section 6.2, it has the added flexibility that the meta-model itself can be used to dynamically extend a given implementation of the platform. In this chapter, one possible such implementation is described, with emphasis on demonstrating the feasibility of the Meta-ORB architecture. In particular, an infrastructure of basic services that are required to support the design is presented in section 6.3. This is followed by a description of the runtime support for the meta-model constructs and the services used for their instantiation, in sections 6.4 and 6.5, respectively. Next, a realisation of the meta-information management framework is presented in section 6.6, in terms of the Type Repository. A concrete design and implementation of the reflection framework, considering its integration with the Type Repository, is then presented in section 6.7. Finally, section 6.8 concludes with the main features and highlights of the work.

6.2 Overall approach and structure

Two levels of abstraction are found in the design presented in Chapters 4 and 5, which enable different degrees of implementation flexibility. Firstly, there are the constructs of the programming model (or meta-model), for which a concrete design must be defined (for their runtime representation) and then hard-coded in a particular implementation of the platform. Secondly, there are the aspects of the platform design that are inherently configurable, and which can be realised in different ways in a particular implementation. This is particularly the case of the platform features that are defined in terms of its own programming model. In this case, the services provided by the platform itself can be used to flexibly define the configuration of such features and, if necessary, the reflection support allows their dynamic reconfiguration.
The abstract design naturally suggests a well-structured implementation, in terms of three main modules, which reflect the organisation of the design as presented in Chapters 4 and 5. These modules are briefly summarised below.

- **The Platform Core** – consists of the features that are necessary to support the Meta-ORB programming model. Specifically, it contains the basic distribution infrastructure, with naming and capsule management services\(^1\) (section 6.3), as well as the core constructs specified in Chapter 4. In addition, the platform core defines the runtime representation for the first-class constructs of the programming model, such as components and interfaces (section 6.4). Finally, higher-level services (section 6.5) are also defined in this module, notably default implementations of component and binding factories.

- **The Type Repository** – this module, described in section 6.6, represents the meta-information management framework of the Meta-ORB, providing support for both the platform core and its meta-level. It mirrors the structure and the functionality defined in the Meta-ORB meta-model, although also introducing tools and mechanisms to facilitate the definition and manipulation of meta-information.

- **The Meta-level** – this module corresponds to the reflection mechanisms and facilities of the platform. It follows the framework defined in Chapter 5, with the design defined in terms of the constructs of the programming model. Thus, the approach used here is to provide a default design and implementation, with meta-object types that offer a representative meta-object protocol. This can later be extended with new meta-object types, either through static type definition, or through reflection (i.e., using meta-meta-objects) and type evolution.

The intent of this implementation is to demonstrate the feasibility of the architecture, through a concrete and complete implementation, which makes it possible to experiment with and evaluate the proposed concepts in realistic application scenarios. Thus, the focus is on the functionality and the qualities of the architecture, instead of performance. This is reflected on the chosen implementation environment, based on the Python programming language [van Rossum 2001], which favours rapid prototyping rather than performance, partly due to its interpretative nature. Despite this, as the experiments discussed in the Chapter 7 demonstrate, the performance of

---

\(^1\) Note that these services have not been specified in the abstract design, as they are not essential for adherence to the meta-model (although their uniform definition is important for interoperability).
Chapter 6 – The Meta-ORB: Concrete Design and Implementation

the prototype is still appropriate for simple multimedia applications. In addition, by implementing the prototype purely in Python, portability to a variety of operating systems is guaranteed, which was also a factor when choosing the language.

Importantly, the implementation of the Meta-ORB in Python requires a language mapping, so that the constructs of the Meta-ORB ODL can be expressed in Python. As the Meta-ORB ODL is an extension of OMG IDL, the basic part of this mapping corresponds to the Python IDL mapping recently standardised by OMG [OMG 2001c], although excluding the features of OMG IDL that were introduced after CORBA 2.2, as well as the mapping for interfaces (which are redefined in the Meta-ORB). The definition of the extended part of the mapping is implied in section 6.4, which describes the structures used to represent the first-class concepts of meta-model.

6.3 Basic infrastructure services

6.3.1 Naming

Overview

A naming service is an essential feature of distributed systems middleware, as entities in the distributed environment need to be identifiable in a unique and unambiguous way [Coulouris et al. 2001]. The Meta-ORB is no exception to this and, although a naming service is not specified as part of the abstract design, its provision is an implicit requirement. In particular, the existence of a scheme for uniquely naming the first-class objects of the meta-model (interfaces, components and binding objects) is assumed in the description of the meta-model and the reflection framework, in Chapters 4 and 5. Note, however, that the treatment given to the subject here is limited to the essential functionality to support the implementation of the Meta-ORB.

Naming scheme

Unique names are defined in a context-sensitive way, using the production rules presented in Figure 6.1. This results in compound names that mirror the composition structure of the respective objects (interfaces, components or bindings), as well as their location in the distributed environment. In this scheme, objects are assumed to
have local names that are unique within their immediate contexts. A naming context can be a component (for interfaces), a composite component (for nested components), a capsule (for top-level components) or a binding (for binding roles and their internal components). The unique name of an object is then obtained by suffixing its local name with the unique name of its context. In addition, it is assumed that the names of top-level bindings are globally unique as they are defined by the user, and that unique names (identifiers) for capsules are attributed by the name server in an automatic way.

```plaintext
//---local names (assigned in type definitions):
<interf_name>, // for interfaces, in the context of a component type
<comp_name>, // for nested components, in the context of a composite type
<binding_name>, // for nested bindings, in the context of a composite binding type
<role_name>  // for binding roles, in the context of a binding type
:: = <string_literal>

//---top-level names:
<capsule_uname>, // globally unique capsule names, generated by the name server
<top_binding_uname>, // globally unique names for top-level bindings
<top_comp_uname>, // locally unique names for top-level components
:: = <string_literal>

//---Interface naming:
<interf_uname> ::= <interf_name> ':' <comp_uname>

//---Component naming:
// top-level components and components defined inside other components:
<comp_uname> ::= <top_comp_uname> ':' <capsule_uname> // top-level components
| <comp_name> ':' <comp_uname> // nested components
| <comp_name> ':' <endp_uname> // components of composite binding objects
| <comp_name> ':' <binding_uname> // control components of binding objects

//---Binding naming:
<binding_uname> ::= <top_binding_uname> // top-level bindings (user-defined unique name)
| <binding_uname> ':' <binding_uname> // nested bindings

//---Binding endpoints:
<endp_uname> ::= <role_name> '@' <capsule_uname> ':' <binding_uname>
```

Figure 6.1 – Naming scheme for the different kinds of objects in the Meta-ORB

Naming service

The name server offers a basic set of operations, shown in Table 6.1, to handle unique names and the binding with their respective objects. The architecture of the name server aims to optimise access, as well as to simplify the use of the service. Considering this, the naming service in a given Meta-ORB domain (defined as the extent of the distributed system under a common administrative authority) is partitioned into one single global name server and a number of local name server proxies, one per capsule. The global name server is responsible for holding the registry of name-object bindings in the domain, and for guaranteeing the uniqueness of names. Proxies in turn are in charge of providing the naming service (i.e., the
operations listed in Table 6.1) to local clients in a capsule, as well as interfacing with the global name server through the network. This facilitates access as otherwise clients would need to use low-level communications primitives (as the high-level programming model of the Meta-ORB does not apply to basic services like the name server). In addition, the name server proxies also cache previously resolved name bindings, in order to speed up future accesses.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register(name, interf)</td>
<td>Registers the name binding between an interface and its unique name, and creates an interface reference for the interface</td>
</tr>
<tr>
<td>register_name(name)</td>
<td>Registers a unique name associated with a non-interface object (e.g., a component or a binding object)</td>
</tr>
<tr>
<td>unregister(name)</td>
<td>De-registers a previously existing name</td>
</tr>
<tr>
<td>lookup(name)</td>
<td>Returns the interface reference associated with the name, if any</td>
</tr>
<tr>
<td>resolve(name)</td>
<td>Returns an active interface reference (i.e., which can be directly used to call the interface, via an implicit binding) associated with the name, if any</td>
</tr>
<tr>
<td>is_unique(name)</td>
<td>Tests the uniqueness of a name (e.g., the name of a top-level binding)</td>
</tr>
<tr>
<td>create_uname( )</td>
<td>Creates a unique name (identifier); mainly used to generate capsule ids</td>
</tr>
</tbody>
</table>

### 6.3.2 Capsules

A **capsule** represents the unit for the location of objects (notably, interfaces and components, as well as the endpoints of binding objects) in the distributed environment. The definition is consistent with RM-ODP, where a capsule is defined as “a configuration of engineering objects forming a single unit for the purpose of encapsulation of processing and storage” [ITU-T/ISO 1995b]. In practice, a capsule in the Meta-ORB corresponds to an operating system process, augmented with data structures and objects for its management and for the provision of infrastructure services. In particular, a capsule contains a **capsule manager**, which keeps a record of the objects contained in the capsule and serves as the access point from which references to the other infrastructure services (such as the naming service and the Type Repository) can be obtained. In addition, a capsule offers the basic communications support for implicit binding, in order that remote clients can directly invoke interfaces located in the capsule (see 6.3.4 for details on this).
6.3.3 Local bindings

The purpose and requirements for local bindings were discussed in Chapter 4. The particular implementation adopted in the prototype is derived from OOPP [Andersen et al. 2000], although in the Meta-ORB a local binding does not exist as a distinct object. Instead, it consists of cross-references between the two bound interfaces, where each interface object has references to the exported interactions (either provided operations, or flows or signals with direction “in”, depending on the interface style) of the other. The method slots that are part of the runtime representation of interfaces (see 6.4.1) are used for establishing such references. In addition, as defined in Chapter 4, the two interfaces connected by a local binding must match each other exactly. This approach has the benefit of simplifying interactions between co-located interfaces, thus contributing to the semantics of instant and reliable interaction via local bindings, as defined in [Blair and Stefani 1997].

6.3.4 Interface references and implicit bindings

The particular implementation of interface references used in the prototype, which also serves as a reference for other implementations of the platform, is described next. The design follows the overall purpose and structure presented in Chapter 4.

Interface references are represented as objects of class IRef, which defines attributes for the identification of the referenced interface (through its unique name and the repository identifier of its type), as well as for the location and direct interaction with the interface. Interface location is represented in the form of the identifier of the capsule where the interface resides. This enables binding factories to locate interfaces as part of explicit binding establishment. For direct interaction in turn, an IRef instance must be augmented with further attributes, as described below.

Interface references are created as part of the process of registering the respective interfaces with the name server. They are then maintained by the name server, from which they can be retrieved using the lookup and resolve operations (Table 6.1). The lookup operation returns a passive interface reference, which can be used to identify and locate an interface, but not to directly invoke it. This is the typical case when obtaining interface references to be used for explicit binding. On the other hand, resolve returns an active interface reference, which can be immediately used to invoke the interface. In other words, resolve effectively enables an implicit binding
to the interface. Note, however, that active interface references (and thus implicit bindings) are only supported for server operational interfaces (where all operations have provided causality). In addition, note that interface references are always created (and stored in the global name server) in passive form. Subsequently, as a result of invoking resolve, the interface reference is turned into the active form (by the name server proxy). The support for active interface references is discussed next.

An active interface reference acts as a remote proxy (or stub) for its interface, offering methods that represent the provided operations defined in the interface’s type. These stub methods are logically defined according to the mapping for operations as specified in the Python IDL mapping [OMG 2001c], and their implementation is responsible for forwarding calls to the actual interface (as described below). In addition, the interface reference also handles the eventual return values from the invoked operation.

As interface references are activated dynamically, Python’s ability to define callable objects that emulate normal methods is used to install the appropriate stub methods in the interface reference object. In this way, a generic function is used as the implementation of the special method __call__ of the callable object, which gets called when the object is invoked. This function handles invocations in a way that is independent of the particular operation definitions, by explicitly representing the elements of operation signatures (notably, operation name, arguments and result).

The actual forwarding of requests, in turn, is done through a low-level communications support, currently based on TCP sockets. An interface object (see 6.4.1) has an associated communications server, which can be either private or shared with other interfaces in the same capsule. This server has a separate thread that runs a loop to wait, receive, unmarshal and deliver messages to the appropriate methods on the interface, as well as to marshal and send eventual results back. In addition, active interface references have associated objects acting as communications clients, which perform the marshalling and unmarshalling of requests and results, and contain the necessary protocol information for sending and receiving them to/from the communications server associated with the referenced interface. Note, however, that an optimised active interface reference is used when client (which invoked resolve) and referenced interface are co-located. In this case, the communications support is bypassed and local method calls are instead used for request forwarding.
6.4 Supporting the programming model

6.4.1 Interfaces

The interfaces of component and binding objects are represented as distinct runtime objects (referred to as interface objects or, simply, interfaces). This allows their direct manipulation, notably for performing local and implicit bindings. In addition, it introduces a level of indirection in the data path, which enables explicit control over individual interactions (e.g., for the installation of pre- and post-methods). This also enables active interfaces (see below), which have their own thread of control, as in [Williams and Arnold 1997]. Note that the overhead of such indirection is usually small, as an interface and its component are always in the same capsule.

Importantly, interface objects do not exist in isolation, as they can only be created as part of components. Furthermore, due to the approach to define the interfaces of composite components and binding objects (which involves the exposure or mapping of appropriate existing interfaces of internal components), interface objects are only created as part of primitive components.

Mirroring the definition of interfaces in the meta-model, there are three kinds of interface objects: operational, stream and signal. However, all these three kinds inherit from a single base class (BaseInterface), which defines the core features that are common to all interfaces. In particular, this base class defines the context-specific name of the represented interface (as defined in the type of the component or binding object that “owns” the interface), its unique name and the repository identifier of its type (which represents the InstanceOf link between an object and its type). Interactions provided through the interface (irrespective of the interface style) are also represented in this base class, in terms of methods for the forwarding of received messages to the actual implementation of the interface. For this purpose, the interface object also contains a reference to the Python object that implements the features of the interface. This object coincides with the object that implements the most primitive component that owns the interface. The forwarding methods are defined according to the Python ODL mapping, and are implemented using callable objects, in a similar way as for active interface references. Note, however, that such methods should not be directly invoked. Instead, they must be invoked either via a local binding or though an interface reference (i.e., via an implicit binding). Finally, the interface object also
represents the interactions that are required by the interface (when it is locally bound to another interface). These are represented in terms of appropriately defined methods (also in conformance with the Python ODL mapping) that enable the forwarding of interactions to the other, locally bound interface. These methods, however, are only created when a local binding to the interface is established (see 6.3.3). Before local binding, they are simply represented by their names in a dictionary data structure, as if they were slots for the actual methods in the interface object.

The creation of an interface object involves type checking of the interface’s type against the implementation object (to make sure that the object supports all the features of the interface), as well as the creation of a forwarding method for each of the interface’s provided interactions. In addition, the interface is also registered with the name server, which enables references to the interface to be obtained through name resolution. Interface creation is performed under the control of a component factory, when creating the component that owns the interface. However, the actual workings are defined in the three classes derived from BaseInterface, representing the three interface styles: OpInterface, SigInterface and StrInterface. These classes do not introduce new features to the runtime representation of interfaces (except in the case of stream interfaces, as seen below). Instead, constructor methods are defined, which are responsible for initialising the interface object appropriately. In particular, the constructor assigns the interactions defined in the interface’s type with the appropriate forwarding methods, as described below.

- **Operational interfaces.** The constructor for the OpInterface class creates one forwarding method for each provided operation declared in the interface’s type and defines one method slot for each required operation. The particular representation for operations is derived from the standard Python-IDL mapping [OMG 2001c]. Thus, operation parameters with “in” or “inout” causality are represented as arguments to the method, while “out” and “inout” parameters, together with the operation result, are represented as the method’s return value (using a tuple if more than one value is to be returned).

- **Signal interfaces.** The constructor for SigInterface creates a forwarding method for each signal with direction “in” and a method slot for each signal with direction “out”. The values of a signal are represented as arguments to the corresponding method (and this method must have no return value).
**Stream interfaces.** Similarly, the constructor for `StrInterface` creates a forwarding method for each flow with direction “in”, as well as a method slot for each flow with direction “out”. A method representing a flow has a single argument and no result value. The argument is used to carry the flow data (e.g., in terms of the individual packets that compose the flow stream). In addition, `StrInterface` has to support an active interface mechanism, for cases where the flow of data through the interface must be controlled by the interface itself. This includes extra initialisation steps, to register call-back methods for the “out” flows (enabling the interface to fetch flow data from its implementation), and to spawn the interface’s control thread, which runs a loop to periodically poll the call-backs and feed the “out” flows. This feature, however, may not be required, in case the interface is passive (i.e., when the flow of data through the interface is controlled by the interface’s implementation). The definition of the mode of the interface (active or passive) is done by explicitly invoking special methods (`set_active` and `set_passive`) on the interface object.

Note that a Python mapping for QoS annotations associated with interactions, as well as for media types, is not currently defined and is a subject for future work.

### 6.4.2 Components

Similarly to interfaces, components are also represented by distinct runtime objects (called *component objects*), in order to enable their direct manipulation. The properties that are common to primitive and composite components are defined in the base class for component objects, `BaseComponent`. These properties are: the context-specific name of the represented component, its unique name, the repository identifier of its type (representing the `InstanceOf` link), a reference to the manager of the capsule where it is located, and a reference to the object within which it is contained (called its `context`), if it is a nested component. In addition, a component object has a list of references to the runtime objects that represent the interfaces it supports.

For *primitive components*, defined by the derived class `PrimComponent`, the runtime representation also contains a reference to the user-defined object that implements the functionality of the component. Importantly, the implementation object should have one method for each of the interactions provided by each of the interfaces supported by the component. Such implementation methods must be
defined according to the mapping for interactions defined in 6.4.1. In addition, for components with active interfaces, the implementation object must also support a *call-back* method for each of the “out” flows defined in such interfaces. Call-back methods have the same name of the respective flows, suffixed with “_callback”.

For *composite components*, defined by the derived class `Component`, the component object also contains a list with references to the runtime objects representing each of the nested components that make up the configuration. Note that, although the internal components are connected to each other, the object graph representing such interconnection is not represented as part of the component’s runtime representation (instead, it is derived from the component’s type).

### 6.4.3 Binding objects

Explicit bindings are distributed objects (i.e., their structure can be spread across more than one address space). For this reason, it is not possible to have a single Python object representing the whole binding (since Python, as with most conventional object-oriented languages, does not support distributed objects in this fashion). Hence, bindings are pseudo objects, and their runtime representation must be achieved through some implementation artefact. However, instead of defining a new kind of object for this purpose, this implementation of the Meta-ORB uses the main control interface of a binding as such a representation. More precisely, the component of the binding that supports the main control interface (called the *control component*) is overloaded with the job of representing the properties of the binding object as a whole. In particular, this component is responsible for representing the unique name of the binding object, the repository identifier of the binding type (which represents the `InstanceOf` link), as well as information about the binding’s internal structure:

- references to the nested bindings (through the interface objects that represent their main control interfaces) – this is useful when needing to control such bindings;
- a list of the endpoints of the binding, where each endpoint is represented as a pair of role name and location (given in terms of the identifier of the capsule where the endpoint is) – this is useful when reconstructing the binding’s object graph.

In order to enable access to such (meta-)information, the control component of the binding must support an additional interface, `config_intf`, with operations to get
and set the above meta-information elements. This configuration interface should be known only to the binding owner (usually, the component or user that requested the creation of the binding) and to eventual meta-objects reifying the binding. This is needed to protect the integrity of meta-information about the binding (although currently there is no specific mechanism to control the access to interfaces).

Note that the above representation is only valid for composite bindings. For primitive bindings (which have no control interface) a special, standard component type (with a standard interface, named pbconfig_interf) is defined for this purpose. This component is responsible for maintaining the same meta-information as described above (except for the item referring to nested bindings).

### 6.5 Creating objects: factories

#### 6.5.1 Default component factory service

The instantiation of components is performed through a component factory service, which is a native feature of capsules. By default, each capsule has one Python object representing the local component factory, which supports an interface with operations that clients (both local and remote) can use to request the creation of components (using active interface references). This is possible by making the component factory object turn itself into a full-fledged component after its initialisation. Figure 6.2 presents the types of the component factory component and its interface.

```plaintext
typedef sequence<any> ArgList;

interface CFInterf {
  string new(
    in CORBA::RepositoryId component_def_id,
    in string component_name,
    in string context_uname,
    in boolean fixed_name,
    in ArgList implem_args)
  raises (ORBcoreException);

  void destroy(in string component_uname)
  raises (ORBcoreException);}

primitive component CFcomp{
  implementation: ComponentFactory;
  interfaces: CFInterf CF;
}
```

Figure 6.2 – The standard component and interface types for component factories
In particular, a component factory is embedded into a primitive component of type CFComp. (Although the implementation of this component is specified, it is not actually used, since the component factory is instantiated as a normal Python object and, only then, converted into a component.) This component type supports an interface of type CFIInterf, which provides two operations, named new and destroy, which are used to create and delete components, respectively. The new operation accepts five arguments: the repository identifier of the component type to be instantiated; the name of the new component; the unique name of the object representing the context inside which the component is to be created; a flag to indicate whether the given name is already a unique name (in which case the previous argument is ignored) or must be qualified with the name of the component’s context; and a list with zero or more arguments to be passed to the component’s implementation (although this is only supported when creating primitive components). An exception is raised in case component creation fails.

The algorithm for creating composite components is independent of the particular implementation of component factories and has been described, in an abstract way, in Chapter 4. The instantiation of primitive components, on the other hand, depends on the particular implementation and is typically realised in three steps. First, the Python code implementing the component is retrieved from the component type and dynamically loaded (currently, the code is assumed to be contained as part of the type, in the form of a bytecode-compiled Python program). Then, a Python object corresponding to the component implementation is created, by instantiating a class with the same name, contained in the implementation program. Any eventual implementation arguments given to the factory are passed to the class’ constructor. Finally, interface objects corresponding to each of the interfaces defined in the component type are created (see section 6.4.1 for details on interface creation).

6.5.2 Default binding factory service

Outline

The overall binding framework was described in Chapter 4, in terms of the metatypes used for the configuration of binding objects and the abstract protocol used for binding establishment. Now, a concrete realisation of this framework will be
presented, based on a default binding service that is natively provided by this implementation of the Meta-ORB. The description focuses on the architecture of binding factories, as well as the concrete binding protocol that they follow. This is a basic binding service, however, and possible extensions will be considered in 6.5.3. Crucially, though, the binding factory service described here is generic and capable of handling the creation of bindings of arbitrary types and configurations.

**Binding factories: replication strategy**

Due to the distributed nature of binding objects, their instantiation inherently requires remote actions to create the necessary binding endpoints (which may be dispersed across the distributed system). Thus, a centralised approach to control the binding creation process (i.e., with a single binding factory object) is inappropriate. For this reason, this implementation of the Meta-ORB adopts a distributed approach, based on replicated binding factory objects, called *local binding factories* (LBF), which cooperate with each other in the process of binding establishment. This approach is also proposed in other representative works on explicit binding, notably, the RM-ODP standard for interface references and binding [ITU-T/ISO 1998b], ReTINA [Dang-Tran et al. 1996], and SumoORB [Blair and Stefani 1997].

Essentially, each capsule has a primitive component realising its LBF. The type of this component, shown in Figure 6.3, uses three distinct interface types for the interfaces of local binding factories. This enables a differentiation of the two distinct roles played by the several LBFs involved in a given binding establishment process: *primary LBF* (which receives the binding request and coordinates the whole process) and *secondary LBFs* (which perform the instantiation of each of the individual endpoints of the binding). The binding factory interfaces are described next, followed by the binding protocol, which shows how these interfaces are used and how LBFs cooperate during binding establishment, according to the two roles described above.

| primitive component BFcomp{ |
| implementation: BindingFactory; |
| interfaces: |
| BFInterf BF; |
| BFsecInterf BFsec; |
| BFcollectInterf BFcollect; |
| }; |

**Figure 6.3 – Component type for local binding factories**
Binding factory interfaces

The BF interface, whose type definition is shown in Figure 6.4, represents the public binding factory service, as seen by its clients. It has a single operation, new, which binding initiators use to request binding establishment. This operation is parameterised by the list of interface references representing the interfaces to be bound, the repository identifier of the particular type of binding to be created and the (unique) name to be given to the new binding object. The LBF on which this operation is invoked becomes the primary LBF for the resulting run of the binding protocol.

```java
interface BFInterf{
    BindingCtrl new(in IRefSeq irref_list,
    in CORBA::RepositoryId binding_def_id,
    in string binding_name)
    raises (ORBcoreException);
};
```

Figure 6.4 – Primary interface of local binding factories

In contrast, the other two interfaces, BFsec and BFcollect, are private to the binding service and are used by the several LBFs participating in a binding establishment to cooperate with each other. In practice, the operations defined in their respective interface types correspond to the core of the binding protocol. For a given binding establishment process, the BFsec interface is active for secondary LBFs, while BFcollect is active for the primary LBF.

The BFsec interface, defined in Figure 6.5, is used to coordinate the secondary LBFs in order to create each of the endpoints of a binding. This is achieved via the two oneway operations new_sec and activate_endpoint. The former is used to invoke a secondary LBF to request the instantiation of an endpoint. It accepts the following arguments: the interface reference of the target interface; the repository identifiers of the binding type and of the role definition to be instantiated; the name of the binding being created; a token to identify the result to be returned to the primary LBF in a separate call (as the involved operations are asynchronous); and the interface reference of the caller LBF (used to send back the result). The latter operation, in turn, is used to request each of the involved secondary LBFs to activate their respective endpoints, so that the binding becomes effective. It accepts a single argument, of type BindingResult, with information about the composite binding and its nested bindings, as required to activate the endpoints. In addition, BFsec also contains an operation for the primary LBF to cancel the creation of an endpoint in case of failure.
**Figure 6.5 – Secondary interface of local binding factories (and accessory definitions)**

```c
typedef sequence <octet> ProtData;
struct PrimBindingResult {
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    ProtData finalProtocolData;
};
struct BindingResult {
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    struct NestedBindingResult{
        sequence <BindingResult> bindingResults;
        sequence <PrimBindingResult> primBindingResults;
    } nestedResults;
};
interface BFsecInterf{
    oneway void new_sec(in IRef target_iref,
                       in CORBA::RepositoryId binding_def_id,
                       in CORBA::RepositoryId role_def_id,
                       in string binding_name,
                       in unsigned long token,
                       in IRef init_iref);
    oneway void activate_endpoint(in BindingResult binding_result);
    oneway void cancel_endpoint(in string binding_name,
                                in string role_name);
};
```

**Figure 6.6 – Collect interface of local binding factories (and accessory definitions)**

Finally, the BFcollect interface (Figure 6.6), contains a single oneway operation, `collect_sec_result`, which enables the secondary LBFs to return the results of the respective endpoint instantiations to the primary LBF. This operation accepts a single argument, of type `SecResult`, with information about the particular created endpoint (this is a recursive structure, including information about all binding levels).
The distributed binding protocol

Cooperation between LBFs, through the operations defined in their interfaces, requires a protocol that governs the sequence of interactions involved, so that bindings can be successfully established. The protocol described here is a specialisation of the abstract binding protocol presented in Chapter 4, considering the replication of binding factories and the distinct roles they can play during binding establishment. The protocol is illustrated through a simple example, shown in Figure 6.7. The dashed lines indicate the different capsules involved in the process (i.e., the capsule from which the binding creation was requested and the two other ones where the target interfaces are located). In what follows, the several steps of the protocol are described.

Figure 6.7 – Illustration of the distributed binding protocol

Step 1. The binding initiator invokes the LBF in its local capsule, by calling operation new on the BF interface, with the following arguments: list of target interface references (e.g., [IRefA, IRefB]), repository identifier of the binding type (e.g., “M-ORB:comp.lancs.ac.uk/BindingTypes/AudioBinding:1.0”), and the binding name (e.g., “AudioBinding1”).
**Step 2.** The primary LBF validates the arguments (uniqueness of the binding name, and existence of the binding type), registers the new binding name, locates the target interfaces (based on the given interface references), matches the target interfaces with the appropriate binding roles (as defined in Chapter 4), and checks that the binding rules (cardinality, causal dependencies, etc.) are not violated.

**Step 3.** For each of the target interfaces, the primary LBF asynchronously invokes the respective secondary LBFs, in order to create the corresponding binding endpoints. This is done by calling the `new_sec` operation, with the arguments discussed above. In addition, the primary LBF records each of the invocations to secondary LBFs, indexed by the respective `tokens`, so that it can correctly associate the reply messages (to be received via the `collect_sec_result` operation). Finally, the primary LBF blocks the current thread\(^2\) until all the replies are received (i.e., until step 6 is completed).

**Step 4.** This step corresponds to the *creation of a binding endpoint*, and is carried out by each of the involved secondary LBFs, as a result of the `new_sec` calls issued in step 3. It consists of the validation of arguments, resolution of the given interface reference (in order to obtain the actual `target interface`), and, notably, the creation of the configuration of the respective binding endpoint. This is performed by interpreting the endpoint configuration description, as obtained from the binding type and from the role definition corresponding to the endpoint being created. From this, two kinds of objects are identified that constitute the endpoint configuration: components and (nested) bindings. The components are created with the use of the local component factory. Local bindings are then established between their interfaces, as specified in the object graph that is part of the role definition. Nested bindings, on the other hand, are created by having the secondary LBF *recursively* invoking its `new_sec` operation, thus effectively re-executing this step for the nested binding endpoint. Note, however, that the creation of primitive binding endpoints, which ends the recursion, is performed differently. In this case, the object implementing the primitive binding functionality is instantiated (as with primitive components), and an interface object is created (to represent the primitive binding’s interface at the current endpoint) and locally bound to the primitive

---

\(^2\) A primary LBF is multi-threaded: it borrows the caller’s thread and receives messages on the `BFcollect` interface using a different thread (spawned by the implicit binding underlying support).
binding’s target interface. Finally, after the recursive call returns, the binding’s interface at the current endpoint is defined (by mapping the appropriate interface of an internal component) and locally bound to the target interface.

**Step 5.** The secondary LBFs return the respective endpoint creation results to the primary LBF, by invoking the `collect_sec_result` operation of its BFcollect interface. The returned `SecResult` structure (see Figure 6.6) contains information about all the levels of the endpoint configuration, notably status indication (i.e., success or failure) and, for primitive binding endpoints, data related to the underlying communications protocol (e.g. protocol-specific addresses).

**Step 6.** The primary LBF, after collecting all the replies via its BFcollect interface, analyses them and decides whether the binding establishment was successful or not. In particular, if there are any failed endpoints, the primary LBF checks if they are essential, according to the binding cardinality and dependency rules. If these rules are not violated by the absence of the failed endpoints, the primary LBF proceeds with effecting the binding establishment (by releasing the lock acquired in step 3); otherwise, it interrupts the process, returns a failure indication to its caller, and undoes the previous steps.

**Step 7.** If the binding establishment was successful, the primary LBF sends confirmation messages to the secondary LBFs, via the `activate_endpoint` operation of their BFsec interfaces. Importantly, the argument of this operation contains any information that is necessary to enable the endpoints to work, considering all levels of the binding, as seen in the recursive definition of the BindingResult data structure (Figure 6.5). Currently, however, only the primitive binding level of this data structure is effectively used, notably, to carry protocol-dependent data required for the primitive binding implementation to set up the underlying protocol connection at each endpoint. Nevertheless, the other levels of the data structure may be used in the future, e.g., to carry the results of parameters (such as QoS) negotiated between the endpoints.

Finally, after step 7 is completed, the primary LBF returns the result of binding establishment to the initiator. The result consists of the interface reference of the main control interface of the new binding, which the caller (also known as the binding owner) can use to control the operation of the binding.
It is necessary to observe that all remote interactions between LBFs as part of the binding protocol are realised with the use of implicit bindings. This is essential to avoid the circular need of a further binding service. In addition, although the example used to illustrate the binding protocol involves only two endpoints, the protocol is generic enough for bindings with an arbitrary number of endpoints. Crucially, the distributed nature of the protocol implies a high level of scalability, as the bulk of binding instantiation (the creation of endpoint configurations) is performed in parallel by different LBFs. Additionally, because the protocol is based on the interpretation of the binding type to be instantiated, it is also generic enough for the instantiation of arbitrary binding types. Note that this is facilitated by the existence of a language mapping (from ODL to Python, as defined in section 6.4), which standardises the runtime representation of components, bindings and interfaces.

### 6.5.3 Possible Extensions

The implementations of component and binding factories presented above constitute one possible realisation of the framework described in Chapter 4. More sophisticated factories can be provided which offer a service that is tailored to particular needs. For instance, an extended component factory can be defined which considers resource management as one factor to determine the feasibility of creating a component. Similarly, an extended binding factory could include QoS negotiation and resource allocation as part of binding establishment, such as in [Kramp and Coulson 2000]. In addition, it is possible to define a binding factory service that does not require the client to identify the type of binding to be created. Instead, the factory would negotiate the binding type, taking into account the definition of the interfaces to be bound, such as suggested in [ITU-T/ISO 1998b] and additional high-level binding requirements, such as proposed in [Rafaelsen and Eliassen 2000].

Because of the different ways component and binding factories are defined, the strategies for defining new designs and implementations are different in each case. As component creation is a fundamental feature to support the programming model, the default component factory is created in a special way, using language-level facilities and then wrapping it with a high-level component definition. This avoids the recurring need of a further component creation service to bootstrap the default component factory. New types of component factory are thus typically built on top of the default
one. This can be done in one of two ways. Firstly, a new component type can aggregate the type of the default factory component (e.g., it can be defined as a composite component which has, as one of its internal components, the default component factory). Secondly, a completely new type of component factory can be defined. In both cases, however, the service of the default component factory is used to create the components corresponding to the new factory. Note that this implies the co-existence of distinct component factory services, which may result in conflicts, such as when one of the factories considers the management of resources. In such cases, it may be necessary to make the different component factories aware of each other, for example, by adopting a common resource management policy.

Binding factories, however, are entirely defined in terms of the constructs of the programming model (in particular, components, interface references and implicit binding). This enables substantial flexibility in the definition of such services, although new binding factory types can still be defined by reusing the default type. In addition, the co-existence of multiple distinct binding factory services needs similar considerations as mentioned above for component factories.

Another important possibility is related to the definition of factories as composite components. This would allow the use of reflection mechanisms to adapt the service of existing factories at runtime, or even to dynamically define new types of factories.

### 6.6 Type repository implementation

#### 6.6.1 Outline and alternatives for implementation

The implementation of the Type Repository (also referred to as the Meta-Information Repository) is considerably straightforward, as its functionality reflects exactly the meta-model described in Chapter 4. However, some implementation decisions are required as to what kind of support to use for its implementation. The natural choice is to use MOF-compliant tools, which take the Meta-ORB meta-model definition (in UML or MODL) and automatically generate the repository interfaces and implementation. Another option is to manually implement the repository, by applying a mapping from the MOF into the Meta-ORB meta-model. The implementation of these two options is described, along with the respective requirements and tradeoffs that led to the choice of the second option for the final
prototype. The required support for the co-existence of multiple type versions is also examined, considering the services that are necessary for type evolution. Finally, tools to facilitate type definition and management are also considered.

### 6.6.2 MOF-compliant implementation

This implementation of the repository was achieved with the use of the dMOF suite from DSTC (see Chapter 3) and an earlier version of the prototype. However, because this tool is only available for Java, its use required porting the Meta-ORB prototype to JPython (now called Jython), a Java implementation of Python that allows Java classes to be transparently used from within Python programs [Jython 2001]. In general, the port was straightforward, due to the compatibility between Python and JPython. Performance, however, was penalised, as JPython tends to be significantly slower than the more usual C implementation of Python.\(^3\)

Using dMOF, the MODL description of the Meta-ORB meta-model (see Appendix B) was processed to generate the meta-model repository server and, from it, the IDL interfaces corresponding to the different kinds of meta-model elements. dMOF was also used to automatically generate implementations (MOFlets) for such interfaces. Taken together, the MOFlets represent the core of the type repository, as they implement the essential operations for meta-information management (as defined by the MOF-IDL mapping). In addition, implementations were provided (as separate Java classes, as required by dMOF) for the specific operations defined in the meta-model elements, notably those that follow the style of the CORBA IR (such as the `lookup` and `create` operations of containers). This was simplified by the use of the basic operations for meta-information access that were automatically generated as part of the MOFlets.

Besides the need to port the prototype into JPython, the use of dMOF had the extra disadvantage that it requires a specific CORBA ORB, currently Visibroker™ for Java, in order to access the repository. In this way, the Meta-ORB becomes dependent on a third party ORB for access to meta-information. In order to overcome such dependence, it would be necessary to have a MOF implementation native to the Meta-ORB, i.e., which maps MOF meta-models directly into the Meta-ORB meta-model

\(^3\) Though this depends on the speed of the underlying JVM and on the use or not of JIT compilation.
(using a MOF-to-ODL mapping). However, this issue falls out of scope in this thesis, although it is a potential subject for future work.

### 6.6.3 IR-based implementation

#### Logical structure

The alternative implementation of the Type Repository was based on an Interface Repository for CORBA 2.2 developed in the early stages of this thesis. This early implementation was based on the translation of the IR interfaces (which represent the meta-types of CORBA 2.2) into Python classes, according to the Python IDL mapping proposed by the Fnorb CORBA ORB [Fnorb 2000]\(^4\). The implementation was also influenced by the Fnorb implementation of the IR, regarding the structures used to represent and create *typecodes*. Note, however, that access to the repository is made through Python method calls (instead of using proper interfaces). This results in an implementation that is more efficient, although it also leads to a physically local implementation. Nevertheless, the replicated design described below is used to overcome this limitation and enable a distributed repository service.

This basic IR implementation was then extended with new interface types (presented in Appendix C) corresponding to the meta-model elements introduced by the Meta-ORB. These new interface types were then mapped into the corresponding Python classes, again using the Python IDL mapping. Interestingly, the process of deriving these interface types from the definition of the meta-model elements can be seen as a *mapping* from MOF to IDL, although different from the standard mapping defined in the MOF specification. Instead, the mapping follows the style of repository interfaces defined in the CORBA IR.

#### Repository replication

In order to enable access to the repository by clients from different locations, a replication strategy is adopted. The repository service (which is realised as a set of local Python objects) is replicated in the different capsules of a Meta-ORB domain. Thus, clients always access the local replica of the repository, by using local method calls. Due to the immutability of the types published in the repository (as defined in

---

\(^4\) This mapping was the basis for the Python mapping recently standardised by OMG [OMG 2001c].
Chapter 4), coherence among the multiple repository replicas is naturally ensured with respect to existing types. However, it is necessary to provide a mechanism to synchronise replicas when new types are published (e.g., when a new version of a type is created due to type evolution). This is done by the replica where the new type was created, which propagates the new type definition to the remaining replicas in the domain. Such interactions are currently based on the underlying TCP sockets support, with multiple one-to-one messages (the addresses of the several type repository replicas are obtained using a special facility provided by the name server). However, it is acknowledged that a more sophisticated approach (such as with a reliable multicast protocol) would be necessary to improve guarantees of coherence among the replicas.

Persistency of the Type Repository

The several replicas of the Type Repository hold the replicated repository contents as in-memory objects. A persistent copy of the repository, in turn, is maintained as a file in a standard location in a shared file system. This file contains a serialised version of each of the objects in the repository. The serialisation process is currently done via the “pickling” algorithm of the Python standard module `cPickle`. At initialisation time, the several repository replicas read this file and de-serialise its contents in order to create the repository objects locally. The persistent copy of the repository is typically created by an interactive tool (such as a type editor; see 6.6.5), which is used to define the repository objects and then to serialise and store them in a file. Further modifications to the persistent copy can then be made through the interactive tool or by the repository replicas themselves, in the case of dynamically created types (as a result of type evolution). In the latter case, to avoid inconsistency of the shared file (by multiple concurrent writes), a single master replica is chosen (statically), which performs the update (by re-serialising the whole of the repository’s contents).  

While this scheme is appropriate for reasonably small repositories, for larger repositories, a more sophisticated persistent store is required. An ideal solution would then be to use a database management system, which would allow the persistent copy of the repository to be updated concurrently and without the need to serialise the whole repository each time it is updated. However, such a solution is considered outside of the scope of this thesis.

5 Note that this update is performed at the same time the individual repository replicas are updated.
Discussion

It is important to note that this kind of solution has a few limitations in comparison with the implementation presented earlier. Firstly, it lacks the flexibility of a MOF-compliant implementation with respect to meta-model extensibility, as meta-type definitions are hard-coded into the implementation. Secondly, it is not capable of explicitly representing associations and constraints. Thus, associations are implicitly represented as attributes of repository objects (which reference other repository objects), and constraints are hard-coded in the classes that implement the repository objects. In addition, although this implementation of the repository is closely based on the standard Python IDL mapping, it is not based on language-independent interfaces, thus being accessible only in Python.

Nevertheless, this implementation suits the current needs of the thesis, which essentially requires the ability to manipulate M1-level meta-information. Crucially, it also avoids the extra complexity of depending on Java and on a foreign ORB. It was, therefore, the chosen option for the final prototype. Note, however, that a native MOF-based implementation, on top of the Meta-ORB and Python, would be necessary if meta-types (i.e., M2-level meta-meta-information) also need to be explicitly manipulated.\(^6\) This is the case with some of the future work discussed in Chapter 8.

6.6.4 Support for type versioning

The type repository implementation, as described above, provides the basic support for type versioning, in terms of repository identifiers. A repository identifier uniquely names repository objects, and consists of a character string with the following format, which is derived from the \textit{OMG IDL} format defined in the CORBA specification:

\texttt{“M-ORB:” <rep\_domain> \{"/" <name>\}”” <version\_number>}

The first element, \texttt{“M-ORB”}, indicates that the repository identifier is defined in the context of the Meta-ORB, while \texttt{<rep\_domain>} identifies the particular Meta-ORB domain where the relevant Type Repository is. The sequence of \texttt{<name>} elements then matches the object’s scoped name (thus identifying its location in the repository), while \texttt{<version\_number>} denotes the version of the particular repository object.

\(^6\) Although the current implementation allows read-only access to the structural aspects of meta-types.
As in the standard CORBA IR, however, the provision of version numbers alone is not enough. Specific mechanisms are required for the management of multiple, co-existing versions. In the Meta-ORB prototype, this is done by providing a complementary service for *version management* and by making the type repository aware of this service when creating new type definitions that are versions of existing ones. In particular, the creation of new versions is differentiated from the creation of non-versioned types. Instead of the usual “create” operations, type versions are created using a special operation, defined as part of the Repository interface:

\[
\text{CORBA:}:\text{RepositoryId new\_version(in any type\_description)};
\]

This operation accepts a type description structure (e.g., `ComponentDescription` or `BindingDescription`), generated from the evolved type definition (as held by the relevant architecture meta-object, as seen in section 6.7), and then returns the repository identifier corresponding to the new type version.

A crucial aspect of the creation of type versions is the management of version numbers, which are the only feature distinguishing the repository identifiers of the different versions of the same original type. Such versions thus need to have unique version numbers. This is especially a problem considering that the several replicas of the Type Repository may be used to create independent versions in a concurrent way. In the current prototype, this problem is solved by adopting a centralised approach to the generation of new version numbers. A central *version server* is defined for the domain, which keeps a record of the original types that have been versioned, as well as the latest version number associated to each of them. Thus, a repository replica, before creating a type version, requests a new version number from the server, which updates its record about the type (by incrementing the version number). In this way, the uniqueness of version numbers (and, thus, of repository identifiers) is ensured. This is, however, a simplified solution, as it does not consider more sophisticated version management facilities, such as the handling of major and minor version numbers, and the tracing of version histories. These issues, however, are out of scope in this thesis and are regarded as potential future work.

### 6.6.5 Associated tools

The interfaces of repository objects, notably in the case of the containers, offer the programmatic support for the *creation* and *manipulation* of types. However, it is
necessary to provide more convenient facilities for this purpose. One means to partially achieve this is through a compiler that validates and translates textual type definitions into objects in the repository. Thus, a compiler can be defined for the Meta-ORB ODL language, which uses the interfaces of container objects to create repository objects based on their ODL definition. Another approach is to use *interactive tools*, which enable users to build and handle type definitions interactively.

The second approach was adopted for the current version of the prototype, as it enables the user to visualise the contents of the whole type repository while defining new types, thus helping to reuse existing features. In addition, it has the ability to guide the user through the process of type definition, checking for type correctness and, thus, eliminating the need to master the ODL syntax. An interactive tool, with a graphical user interface, was therefore developed which can work both as a *browser*, for the navigation through the repository's contents, and as an *editor*, for the definition of new types. This tool can be obtained from the Meta-ORB web site [Costa 2001].

### 6.7 Meta-level implementation

#### 6.7.1 Overview

In this section, the concrete design and implementation of the Meta-ORB reflection framework is considered, based on the abstract design described in Chapter 5. In particular, a well-defined *meta-object protocol* is presented which precisely specifies the facilities defined by the *structural meta-space models*. Importantly, as the meta-level is built using the Meta-ORB programming model, meta-objects are defined as components and the MOP is realised by means of the interfaces of such components. Access to the MOP is then typically achieved by means of implicit bindings. In addition, the programming model concepts are also used to realise the link between base- and meta-level (although this also requires the use of language-level access).

Note, however, that the meta-level implementation presented here is a basic, default implementation, which can be extended with new services (i.e., new MOPs) or different implementation strategies (see discussion in 6.7.7). Importantly, because the meta-level is defined in terms of the usual meta-model elements, it can be extended in a dynamic way, without the need to reinitialise the platform.
6.7.2 Meta-object protocols

The definition of the meta-space models presented in Chapter 5 is abstract and leaves open the design of the precise MOPs. The following tables present the default MOPs, which provide representative meta-level operations and constitute an example of how the meta-space models can be made concrete. Note, however, that only the structural meta-space models are developed, as they constitute the focus of the thesis. The precise definition of the operations is given in Appendix C, using IDL syntax. In addition, note that it was not the intention in this thesis to provide a comprehensive or generic MOP for reflective middleware. Instead, the overall approach is to enable extensibility of the meta-level, through the use of the meta-model constructs to define new meta-objects, which can be made to provide custom MOPs.

Table 6.2 – Interface Discovery MOP (interface type: InterfDiscMeta_interf)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_interf_names</td>
<td>returns a list with the unique names of all interfaces of the base-level object (which can be either a component or a binding object)</td>
</tr>
<tr>
<td>get_endpoints</td>
<td>for binding objects only; returns a list with information about all the endpoints (i.e. role instances) of the base-level binding object; endpoint information includes the respective role names and capsule identifiers</td>
</tr>
<tr>
<td>get_target_interf_type_id</td>
<td>for bindings only; returns the repository identifier of the type of target interfaces expected by a binding role</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the type of the base-level component or binding</td>
</tr>
<tr>
<td>replaceable_by</td>
<td>compares the type of the base-level component (binding) with another type, in order to check if a component (binding) of the other type can be a correct substitute for it; substitutability is currently determined by structural compatibility, i.e., if the new component (binding) supports all the interfaces (roles) supported by the original one; if yes, the return is a list with a mapping between the interfaces (roles) of the base-level component (binding) and the names of the interfaces (roles) declared in the substitute type; if not, an empty list is returned7</td>
</tr>
</tbody>
</table>

7 Note, however, that in future versions of the prototype, such functionality should be migrated to the repository objects representing component and binding types
### Table 6.3 – Interface MOP (interface type: InterfMeta_interf)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_interf_name</td>
<td>returns the unique name of the interface</td>
</tr>
<tr>
<td>get_interf_style</td>
<td>returns a string with the interface style (operational, stream or signal)</td>
</tr>
<tr>
<td>get_interf_descr</td>
<td>returns a complete description of the interface</td>
</tr>
<tr>
<td>get_attr_list</td>
<td>returns a list of strings with the names of all attributes of the interface</td>
</tr>
<tr>
<td>get_attr_type</td>
<td>returns the typecode that represents the type of a given attribute</td>
</tr>
<tr>
<td>get_interaction_list</td>
<td>returns a list of strings with the names of all interactions (operations, flows or signals) defined for the interface</td>
</tr>
<tr>
<td>get_prov_interactions</td>
<td>these two operations return a list of strings with the names of interactions offered by the interface, according to their causality; the results must be interpreted according to the interface style (operations, flows or signals)</td>
</tr>
<tr>
<td>get_req_interactions</td>
<td></td>
</tr>
<tr>
<td>get_interaction_desc</td>
<td>returns a structure containing the complete description of a given interaction; the result must be interpreted according to the interface style</td>
</tr>
<tr>
<td>get_attr_value</td>
<td>returns the current value of an attribute</td>
</tr>
<tr>
<td>set_attr_value</td>
<td>modifies the value of an attribute</td>
</tr>
<tr>
<td>invoke</td>
<td>dynamically invokes an operation</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the interface’s type</td>
</tr>
</tbody>
</table>

### Table 6.4 – Architecture MOP (interface type: ArchMeta_interf)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_obj_graph</td>
<td>returns a data structure representing the internal configuration (i.e. the corresponding object graph) of the base-level object</td>
</tr>
<tr>
<td>get_internal_comps</td>
<td>returns a list with the unique names of the internal components</td>
</tr>
<tr>
<td>get_bound_comps</td>
<td>given a component, returns information about the other components bound to it (including the unique names of the components and their respective interfaces)</td>
</tr>
<tr>
<td>get_role_config</td>
<td>for bindings only; returns a data structure representing the abstract configuration associated with the given role, as in the binding type</td>
</tr>
<tr>
<td>get_endp_config</td>
<td>for bindings only; returns a data structure representing the concrete configuration of the given binding endpoint</td>
</tr>
<tr>
<td>get_internal_bindings</td>
<td>for bindings only; returns the unique names of the nested bindings (but only at the current level of composition)</td>
</tr>
</tbody>
</table>
Table 6.4 (cont.) – Architecture MOP (interface type: ArchMeta_interf)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Operations for configuration adaptation (with corresponding updates to the self-representation)</strong></td>
<td></td>
</tr>
<tr>
<td>local_bind</td>
<td>performs a local binding between the interfaces of two components (or a component and a binding endpoint) that are part of the current configuration</td>
</tr>
<tr>
<td>break_local_binding</td>
<td>breaks an existing local binding within the current configuration</td>
</tr>
<tr>
<td>insert_component</td>
<td>inserts a component of a given type and name at a given location in the configuration (the location is expressed as a pair of interfaces between which the component is to be inserted)</td>
</tr>
<tr>
<td>remove_component</td>
<td>removes and deletes a component from the configuration, re-binding the adjacent interfaces when possible (i.e., when the removed component was bound to two other components, the operation tries to bind their interfaces, so that the configuration is not left broken)</td>
</tr>
<tr>
<td>replace_component</td>
<td>replaces a given component in the configuration with a new one; the operation only succeeds if the new component can correctly replace the old one in all local bindings it took part (this is verified by calling operation replaceable_by of the Interface Discovery meta-object); the old component is deleted</td>
</tr>
<tr>
<td>replace_binding</td>
<td>for bindings only; replaces a nested binding, with the requirement that the new binding must support all the roles of the replaced one; the configuration of the old binding is deleted</td>
</tr>
<tr>
<td>add_component</td>
<td>for components only; adds a new, non-connected component to the configuration of the base-level component</td>
</tr>
<tr>
<td>add_binding_component</td>
<td>for bindings only; adds a new, non-connected component to a particular endpoint of the base-level composite binding object</td>
</tr>
<tr>
<td>role_add_component</td>
<td>for bindings only; these operations act like their counterparts defined above (operations with the same name, except for the ‘role_’ prefix), but applied to the abstract configuration of a given binding role, and with their effects reproduced on all endpoints that are instances of this role in the base-level binding object</td>
</tr>
<tr>
<td>role_break_bind</td>
<td></td>
</tr>
<tr>
<td>role_break_local_binding</td>
<td></td>
</tr>
<tr>
<td>role_insert_component</td>
<td></td>
</tr>
<tr>
<td>role_remove_component</td>
<td></td>
</tr>
<tr>
<td>role_replace_component</td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous operations</strong></td>
<td></td>
</tr>
<tr>
<td>get_interf_exposer</td>
<td>returns the unique name of the internal component exposing a particular interface of the base-level component (or binding object); useful, e.g., when replacing (part of) the implementation behind a given interface</td>
</tr>
<tr>
<td>expose_interf</td>
<td>maps a particular interface of a given internal component as an external interface of the overall composite</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the type of the base-level object</td>
</tr>
<tr>
<td>get_type_descr</td>
<td>returns an up-to-date description of the type (either as a ComponentDescription or as a BindingDescription structure)</td>
</tr>
<tr>
<td>commit_type</td>
<td>used for type evolution; when the copy of the type held by the meta-object has been changed, this operation can be called in order to publish the changed type as a new type version in the Type Repository</td>
</tr>
</tbody>
</table>
6.7.3 Meta-object types and their implementation

Outline

The default meta-objects for the structural meta-space models are defined as primitive components, implying that their implementation must be provided directly in terms of appropriate Python classes. The respective component and interface types, which provide the MOPs described in 6.7.2, are presented in Appendix C. The corresponding Python-based implementations, in turn, can be obtained from the Meta-ORB web site [Costa 2001].

In general terms, the implementation classes of meta-objects have to provide methods to support each of the operations of the corresponding MOP. The signatures of these methods are defined according to the Python IDL mapping [OMG 2001c]. In addition, the implementation classes have to support the meta-level mechanisms defined in Chapter 5, notably meta-object initialisation, type caching and type evolution. In what follows, the most important features of the meta-objects and their implementations are considered, along with the issue of interference between them.

Interface Discovery meta-objects

These meta-objects are defined as components of type InterfDiscMeta_comp. The implementation is simple, as this meta-space model only allows introspection. In most cases, the meta-object simply serves as a convenient way to access meta-information defined in the base-level object’s type (although with the advantage that such meta-information will always be up-to-date, especially in case of type evolution). For this purpose, the meta-object maintains a cached copy of the type (as a type description structure). However, in the case of the get_endpoints operation, additional meta-information must be obtained from the runtime representation of the base-level (binding) object. More precisely, the configuration interface (config_interf) of the binding’s control component is used for this purpose, via its get_endpoints operation, which returns a list of pairs <role, capsule> naming the particular role instantiated at each endpoint, as well as the capsule where the endpoint is located. In addition, in the case of the get_interf_names operation, extra runtime meta-information (the unique name of the base-level object) is also required to determine the unique names of the interfaces. Importantly, some operations of the
MOP are only valid if the base-level object is a binding. If these operations are invoked in other circumstances, an exception is raised.

Interface meta-objects

Interface meta-objects, in turn, are defined as primitive components of type `InterfMeta_comp`. Similarly to Interface Discovery meta-objects, they are a convenient way to access meta-information defined in the type of the base-level interface (which is also cached by the meta-object). An exception is the case of the `get_interf_name` operation, which gets the unique name from the runtime object representing the base-level interface. In addition, the dynamic access operations require direct access to the implementation of an interface (achieved via a reference to the implementation object and by using Python’s `eval` and `apply` built-in functions).

Architecture meta-objects

In contrast to meta-objects of the two other meta-space models, the implementation of Architecture meta-objects is rather complex, due to its double role of introspection and adaptation of component and binding configurations. In addition, the meta-object must maintain a considerable amount of meta-information, part of it not directly found in the base-level object’s type. Hence, this meta-information has to be derived during meta-object initialisation, based on the type of the base-level object and on extra meta-information obtained from the runtime representation of the base-level object. In particular, if the base-level object is a `component`, this meta-information consists of the unique names of the internal components (obtained from the runtime), as well as the object graph and the mapping for external interfaces (obtained from the type).

For base-level binding objects, in turn, this meta-information is partly obtained from the configuration interface of the binding’s control component, notably information about the actual binding endpoints (role name and location). This is then used to derive the meta-information that represents the concrete configuration of the base-level object, which consists of the unique names of the internal components and nested bindings, as well as the concrete object graph. Interestingly, the unique names are determined using the same naming scheme applied by the binding factory (see Figure 6.1). In addition, the meta-object also maintains the descriptions of the binding roles and the mapping for control interfaces, both directly obtained from the type.
Importantly, the meta-information maintained by the meta-object becomes the local (cached) representation of the base-level object’s type. Note that, although the type is not represented explicitly, its type description can be easily derived from the available meta-information (using the reverse process as used during meta-object initialisation).

The operations of the Architecture MOP (see Table 6.4) are then implemented based on such meta-information. In particular, the introspection operations (such as get_obj_graph and get_role_config) are convenient ways to access this meta-information. The operations for adaptation, in contrast, such as insert_component and role_add_component, work by modifying such meta-information (e.g., by changing the object graph or the configuration of a role). Crucially, though, the effects of such changes have to be reflected in the actual configuration of the base-level object. This constitutes the causal-connection link, which is examined in 6.7.5 below.

In addition, adaptation operations that alter role descriptions (i.e., those prefixed with “role_” in Table 6.4), as well as operation expose_interf (which changes the set of interfaces supported by the base-level object), imply the evolution of the base-level object’s type. Consequently, the meta-information maintained by the meta-object no longer corresponds to the original type in the Repository. Subsequent accesses to the type must therefore be made through the meta-object (via the inspection operations), from which up-to-date type meta-information can be obtained. In addition, the Architecture meta-object also offers support for publishing the new type of its base-level object. This will be discussed in 6.7.6 below.

Interference between meta-objects

As seen in Chapter 5, this is only an issue when considering the Architecture and the Interface Discovery meta-space models, in particular, when the former is used to add a new interface to the base-level object. In the current implementation, the solution is to require the Architecture meta-object, after adding a new interface, to update a special data structure in the runtime object representing its base-level. The Interface Discovery meta-object then relies on this data structure to contain the latest description of the base-level object’s external features. Interestingly, this same approach may be used for other meta-space model dependencies introduced in the future. In addition, the use of the runtime representation as the store for meta-
information elements shared by meta-objects avoids the need for complex notification mechanisms between meta-objects.

Interference with the base-level

Reflective adaptations may have side effects on the functioning of base-level objects, particularly in the case of architectural adaptations, which may interfere with the flow of interactions through the base-level object. Such effects may range from temporary disruption of the data path to interaction inconsistencies (e.g., during adaptation, residual interactions may still rely on protocols used by the old configuration) and loss of information (e.g., when the state of a replaced component is not properly transferred to the new component).

Currently, the implementation partially tackles this issue by requiring that a configuration is not operational during its adaptation. As this is particularly an issue when adapting bindings, it is assumed that binding control interfaces (which are user-defined) have stop and start operations, which must be invoked immediately before and after adaptations. In addition, it is assumed that the components of a binding are stateless. However, more complete solutions may be required to achieve adaptation smoothness, such as using the ‘hot swap’ techniques and compound adaptations considered in [Fitzpatrick 2000]. This is a potential subject for future work.

6.7.4 The basic MOP: creating and accessing meta-objects

The platform infrastructure provides a set of functions for access to the meta-space of a given base-level object, as defined in Figure 6.8 below.

```
// For the Interface meta-object:
ORBcore::IRef get_interf_mobj(in string interf_uname)
  raises (Reflection::ReflectionException);

// For the Interface Discovery meta-object (argument may be the unique
// name of a component or a binding object):
ORBcore::IRef get_interf_disc_mobj(in string obj_uname)
  raises (Reflection::ReflectionException);

// For the Architecture meta-object (argument may be the unique name of
// a composite component or binding object):
ORBcore::IRef get_arch_mobj(in string obj_uname)
  raises (Reflection::ReflectionException);
```

Figure 6.8 – Operations of the basic MOP

When invoked, with the unique name of the base-level object as the argument, these functions return an active interface reference to the meta-object corresponding to
their respective meta-space models. The returned interface reference can then be immediately used to invoke the meta-object. In each case, the operation raises an exception if the argument does not refer to an object of the kind reified by the corresponding meta-space model. A special case is when get_arch_mobj is invoked with an argument corresponding to a primitive component or binding; in this case, a null value is returned, indicating that the object has no architecture to be reified.

The underlying working of these functions depends on whether the required meta-object already exists or not. In the former case, an interface reference to the meta-object’s interface must be retrieved and then returned to the caller. This is possible with the use of a meta-object service, which consists of a single server running in a Meta-ORB domain. This server accepts meta-object registrations, whereby the interface reference of the meta-object is recorded, along with the unique name of the respective base-level object and the name of the corresponding meta-space model. In addition, the meta-object server also accepts queries for meta-objects, based on the unique name of the base-level object and on the name of the meta-space model. If the meta-object exists, its interface reference is returned; otherwise, a null value is returned. Thus, the basic MOP functions always call the meta-object server in order to determine if the meta-object already exists or needs to be created.

In the latter case, the component representing the meta-object is created, by using the local component factory, and then registered with the meta-object server for future accesses (before returning its interface reference to the caller). Importantly, a meta-object is always created in the same capsule where its base-level object is located (if the base-level object is a binding, the meta-object is co-located with the binding’s control component). This is so in order to optimise the causal-connection link, discussed next. In addition, because meta-objects are created as components, with well-defined interfaces, they can be accessed by clients in a location-transparent way.

6.7.5 The causal-connection link

The link between base- and meta-level (referred to in Chapter 5 as the Base-Meta link) enables base- and meta-objects to access each other. As the meta-model does not enable this link to be represented explicitly, it has to be realised in an ad hoc way in the implementation of base- and meta-objects. In particular, a meta-object’s implementation maintains a language-level reference to the runtime object that
represents its base-level. This is possible because, as seen above, object and meta-object are always co-located. In addition, a base-level object may maintain interface references to the meta-objects that have been created for it (alternatively, these interface references can be obtained via the basic MOP whenever they are needed).

These two references are essential for the reflection process in general, and for causal connection in particular. The interface reference to the meta-object allows the base-level object (or indeed any client of the meta-object) to request reflective computation. The reference to the base-level runtime object, in turn, enables the meta-object to reflect the results of adaptations in the object’s structure, by directly manipulating its runtime representation. This characterises the reflection (absorption) action, described in Chapter 2. Figure 6.9 illustrates the direction of this action.

![Causal Connection Link](image)

**Figure 6.9 – The causal-connection link**

The figure also shows the reification action, which results in the meta-object building or updating the meta-representation of its base-level object. In the current design, however, the reification action is only supported during meta-object initialisation. Crucially, this implies that base-level objects can only be changed by means of the MOP, instead of directly using the infrastructure services (e.g., local bindings or component creation) for this purpose. If this assumption holds, there is no need for meta-objects to implicitly update the meta-representation, and causal-connection is automatically ensured. Updating reification actions, on the other hand, would require additional mechanisms, such as event notification (to inform a meta-object of implicit adaptations), which are not currently defined in this implementation.

### 6.7.6 Type evolution issues

As proposed in Chapter 5, one possible consequence of reflective adaptation (via the Architecture meta-space model) is the evolution of the type of the base-level object. The evolved type is initially maintained privately by the meta-object, although it can also be published in the Type Repository for subsequent reuse. This is achieved
through the `commit_type` operation of the Architecture MOP, which in turn invokes the `new_version` operation of the Type Repository. Importantly, the `commit_type` operation generates a *description* of the new type (either a `ComponentDescription` or a `BindingDescription`), which is then passed as the argument to `new_version`. The type description is derived from the elements of meta-information maintained by the meta-object, notably the object graph and external interfaces (for component base-level objects), or role descriptions, nested bindings and control interfaces (for binding base-level objects). In any case, evolving the type of the base-level object requires the `InstanceOf` link (represented by the repository identifier maintained by the base-level object’s runtime representation) to be redirected, in order to point to the new type.

Another related issue is the use of *type evolution constraints*, which can be imposed on type-changing reflective operations, so that inconsistent new types (according to some adopted criteria) are avoided. Currently, however, flexible support to such constraints is not provided, and the only constraint supported, structural sub-typing, is hard-coded in the implementation. More precisely, this constraint is fixed in the definition of the Architecture MOP, which does not provide operations that would change the base-level object in a way that would violate structural sub-typing (e.g., it is not possible to remove an interface of a component). Arguably, this is a simplistic approach, and a flexible support to such constraints is considered as future work.

### 6.7.7 Further considerations

The implementation of the meta-level presented in this section, although complete, has some limitations, which opens the possibility for improvements. In particular, the definition of meta-objects as primitive components prevents the use of reflection to adapt the meta-level itself (although reflection can still be used for its introspection). In contrast, if meta-objects were defined as composite components, it would be possible to use Architecture *meta-meta-objects* to adapt their internal configuration (thus changing the kind of service they provide) and to add new interfaces with a different MOP. Importantly, it would be possible to dynamically generate new meta-object types via type evolution. However, this raises the risk of improper uses of reflection, since the consequences may spread through all base-level objects that are reified by meta-objects of an evolved type. Extra security measures (such as access
control to interfaces) are thus required which are not currently part of the prototype. Hence, it was decided to avoid such adaptations, at least for the default meta-objects.

Nevertheless, the design of the meta-level is essentially flexible, in the sense that new meta-object types can be defined, albeit statically, which provide different MOPs or implementation strategies and enables the above-mentioned meta-level adaptations. Interestingly, new meta-object types may be defined by reusing the implementation of the default ones, such as by inheriting their implementations or by aggregating their component types as internal components of more sophisticated meta-object types. In addition, a newly defined meta-object type can be published in the Type Repository and immediately used to instantiate new meta-objects. Note, however, that to fully support this, basic MOP operations have to be extended with an extra, optional parameter, to enable the identification of the type of meta-object to be created.

6.8 Summary

An implementation of the Meta-ORB architecture was presented in this chapter. The description follows the approach defined in Chapters 4 and 5, whereby the meta-model offers the constructs that are used to define both platform and application configurations. Hence, support for the meta-model was described first, including the infrastructure services and the runtime representation for its constructs. The implementation of a meta-information repository based on the meta-model was then described, which enables the definition and management of the types of entities used in configurations. Finally, an implementation of the meta-level was presented, in terms of components that realise the design of the structural meta-space models.

Note that the implementation of the platform does not imply a particular structure or configuration. Instead, it provides the facilities that are required for the flexible definition and instantiation of such configurations. This is complemented by the meta-level, with a well-defined MOP that enables the inspection and adaptation of platform and application configurations. Importantly, the implementation of the platform itself is extensible, notably the factory services and the meta-level.

The use of the features described in this chapter is discussed in Chapter 7, in terms of a case study used to evaluate the Meta-ORB architecture and programming model. The complete prototype implementation, in turn, including a brief installation and user’s guide, can be obtained from the Meta-ORB web site [Costa 2001].
Chapter 7  Evaluation

7.1 Introduction

This chapter presents an evaluation of the Meta-ORB reflective middleware architecture. The overall methodology used in the evaluation is discussed in section 7.2. Section 7.3 then presents a quantitative evaluation of the performance-related aspects of the architecture, based on its current implementation. Following this, section 7.4 presents an evaluation of the architecture and its programming model from a qualitative point of view. Finally, section 7.5 summarises the main conclusions.

7.2 Methods of evaluation

As a middleware platform, the Meta-ORB must be evaluated both from a quantitative and from a qualitative perspective, in order to enable the analysis of its performance-related aspects, as well as the more abstract properties of its programming model.

The main aim of the quantitative evaluation is to analyse the efficiency of the meta-model, considering issues of scalability, overhead and performance estimation. The motivation for this study comes from the fact that the meta-model is the basis for other major techniques used in the platform, notably those for configurability and reflection. Although absolute performance is not a central goal of the work, it is important to demonstrate that these techniques can perform well and scale. The main emphasis is therefore on the performance impact of the several meta-model constructs, providing a basis to estimate the overall performance of such techniques. Nevertheless, in order to complement and illustrate the analysis, concrete performance figures will also be presented, based on the current prototype. In addition, to make such figures meaningful, whenever applicable, they will be compared with similar results obtained with representative baseline platforms in the same implementation environment, as well as with the requirements of simple multimedia applications. This will provide a preliminary view concerning the level of performance that can be achieved with the Meta-ORB, as well as regarding its suitability for one of its main areas of application.
The qualitative evaluation, in turn, examines the properties of the architecture that cannot be meaningfully quantified, notably related to configurability and dynamic reconfigurability. The analysis will be based on a case study, which will enable to demonstrate and examine the use of the several aspects of the architecture in a representative application scenario. The aim is to allow conclusions about the qualities of the architecture and its programming model, notably regarding usability, power of expression and the impact on the development of distributed applications. The study will also enable the identification of further enhancements to improve such qualities.

In addition to the evaluation presented in this chapter, a further assessment of the approach proposed in this thesis is presented in Chapter 8, comparing the features of the Meta-ORB with the reflective middleware requirements introduced in Chapter 2.

7.3 Quantitative evaluation

7.3.1 Overview

This section analyses the performance of the establishment, interaction and reflection aspects of the Meta-ORB. In the first case, the study considers the creation of local bindings, interfaces, components, and binding objects. The study also considers access to the type repository, which is used by most of the above operations. In addition, expressions will be presented which can be used to estimate the execution times of complex establishment operations in terms of their more primitive elements. This helps to identify the critical factors affecting the overall performance of platform configurations. In addition, the study also supports the analysis of the performance of reflection, which in turn will concentrate on the execution times of reification and adaptation actions.

In the case of the interaction mechanisms, the delay and throughput of local and distributed bindings will be analysed. For the latter, however, only minimal configurations are considered, as the performance of more complex configurations results from the composition of the several local bindings and components in the data path. The achieved results will be compared with Python method calls (for local bindings) and with TCP/IP sockets and the Fnorb CORBA ORB [Fnorb 2000] (for binding objects), in order to assess the overhead introduced by the meta-model.
All experiments were conducted on a 10Mbps Ethernet LAN, using identical Pentium III 800MHz PCs with 256MB RAM, running Windows 2000 and Python 2.1. The experiments used the default component and binding factories described in Chapter 6. All measurements were taken using the clock function of the standard Python module time (which is the standard for measuring the performance of Python programs) and averages were used in order to smooth the effects of the non-determinism introduced by the underlying operating system scheduler and network.

7.3.2 Establishment performance

Type repository access

The instantiation of important meta-model constructs requires access to the relevant types in the repository, thus demanding a brief analysis of the associated performance. In general, the time required for type access depends on the particular search operation used. For lookup_id, which is used by the establishment mechanisms, access time is proportional to the length (number of simple names) of the repository identifier of the searched type. To illustrate, for an identifier of length four, say “M-ORB:comp.lancs.ac.uk/Examples/Interfaces/Stream/SimpleAudio:1.0”, access time is around 380μs. Typically, the time increases with the length of the repository identifier by a factor of 70μs, which corresponds mainly to the time for accessing a Python dictionary representing the contents of each container visited during the search. A similar analysis and times are also valid for the lookup operation.

Interface instantiation

The performance of interface instantiation depends on the number and complexity of the interactions defined in the respective interface type. This is notably due to the need for type-checking the interface implementation with respect to the interactions that the interface provides. However, as the current implementation of the Meta-ORB only checks the interaction names (thus not performing type checking of parameters, return values, signal values and flow types), the analysis only considers the number of interactions. Thus, the same results apply to operational, stream and signal interfaces. The elements involved in the performance of interface instantiation, along with their respective times, are shown in Table 7.1.
Table 7.1 – Interface instantiation time

<table>
<thead>
<tr>
<th>Basic interface instantiation time (remote to name server)</th>
<th>2880µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average increment for each extra <em>provided</em> interaction</td>
<td>50µs</td>
</tr>
<tr>
<td>Average increment for each extra <em>required</em> interaction</td>
<td>26µs</td>
</tr>
</tbody>
</table>

The basic instantiation time corresponds to the special case of an interface with no interactions. This time is dominated by the overhead of registering the interface name with the name server, which is about 2800µs (2250µs if local to the name server). This suggests that name server optimisation is the most effective way to reduce interface instantiation time. Starting with this basic time, the total interface instantiation time grows linearly with the number of interactions in the interface type (attribute access operations have the same impact as provided interactions). Note that the time added by *provided interactions* is higher due to the need for type checking (discussed above) and for creating the respective forwarding methods in the interface object (see Chapter 6). For *required interactions*, in contrast, the only processing involved is for preparing the method slots for future local bindings (see below).\(^1\)

Local binding establishment and breaking

The time for *local binding establishment* depends on the number of matching interactions in the involved interfaces (e.g., pairs of “in” and “out” flows). For each pair of matching interactions, the process involves the creation of a callable reference from the requiring to the provider interface, consisting of a forwarding method object in the respective method slot (see Chapter 6). Note that the time for checking the interfaces for compatibility is not significant, as only the interaction names (maintained in a Python dictionary in the interface objects) are verified, instead of full type checking.\(^2\) With this in mind, for interfaces with one pair of matching interactions local binding time is around 140µs, with each extra pair adding, on average, 15µs.

On the other hand, the impact of the number of interactions on the time for *breaking local bindings* is much less significant (as the process simply involves the

---

\(^1\) Note that the analysis does not consider active (stream) interfaces, which would add an extra overhead to the instantiation of interfaces with “out” flows (e.g., to register the callback methods).

\(^2\) This assumes that local bindings are part of larger configurations, which have been fully type-checked during type definition. Note that, although this does not hold for dynamically established local bindings, appropriate type checking in this case is performed if such bindings are done via reflection.
deletion of the cross-references between the two interfaces). For one interaction pair, the time is around 110µs, adding on average 2µs for each extra pair.

Component instantiation

The instantiation time for primitive components is intrinsically dependent on the number and complexity of the component's interfaces. In Figure 7.1 these times are shown for primitive components with up to 10 interfaces (each with a single interaction). The formula below shows how the instantiation time can be estimated in terms of the more elementary operations involved:

\[
prim\_comp\_inst \equiv type\_access + comp\_obj\_inst + impl\_load +
\sum_{i=1}^{n\_ifaces} iface\_type\_access + n\_ifaces \times iface\_inst_i
\] (1)

![Figure 7.1 – Component instantiation times](image)

The basic instantiation time is shown by the special case of a component with no interfaces (around 1ms in this particular experiment). This consists of the time to access the component type in the repository, to instantiate the runtime representation of the component, and to load its implementation. Note that the latter will vary, depending on the size of the component implementation and the amount of processing required for its initialisation (the experiments used dummy implementations with minimal size and near-zero initialisation time). The remaining instantiation time is represented by the time required to access the type of each of the component’s interfaces in the repository (which is constant, as the component type has direct references to such types), plus the time to instantiate these interfaces (see Table 7.1).
For composite components in turn, two illustrative experiments are shown, based on a succession of configurations with increasing numbers of nested components and local bindings. In the first one, the compositions are linear and flat, consisting of a pipeline of up to 10 nested primitive components (each one with two interfaces of a single interaction each) connected by local bindings. The experiment shows that this kind of composition does not inherently add a significant overhead, as the increment in time from one component to the next in the sequence roughly equals the time to create the extra nested component (around 7ms) and local binding (around 0.14ms).

In the second experiment, an illustrative recursive composition was used, with a series of components as shown in Figure 7.2 below. As expected (again see Figure 7.1), the use of nested composite components introduces an extra overhead in comparison with the previous experiment (note that in both cases, the number of primitive components and local bindings is the same). This overhead results from the creation of the runtime objects representing each of the nested composite components, and also from the cost to map their interfaces (incurred mainly by name registration). Nevertheless, the overhead of nested compositions (as long as well managed) is usually a reasonable price to pay for clearer designs and increased reusability.

![Figure 7.2 - Recursive composition experiment](image)

In order to generalise the analysis the formula below shows the decomposition of composite component instantiation time into its more primitive elements:

\[
\text{comp\_inst} \equiv \text{type\_access} + \text{comp\_obj\_inst} + \left( \sum_{i=1}^{n_{\text{prim\_comps}}} \text{prim\_comp\_inst}_i \right) + \\
\left( \sum_{j=1}^{n_{\text{comps}}} \text{comp\_inst}_j \right) + \left( \sum_{k=1}^{n_{\text{binds}}} \text{local\_bind}_k \right) + (n_{\text{ifaces}} \times \text{iface\_map})
\]

In particular, the instantiation time depends on the number of primitive and composite nested components, plus the number of local bindings involved in the configuration. Apart from this, there is the basic time to access the component type and to create the component’s runtime representation, as well as the time to expose its interfaces.
Composite binding establishment

This experiment illustrates the performance and the scalability of the binding protocol for multi-point bindings with up to six endpoints, each one located in a different machine (so that the local binding factories, called LBFs hereafter, do not interfere with each other’s performance). In order to highlight the properties of the protocol, a binding type with minimally configured endpoints is used, each one consisting of a stub (a primitive component) and the endpoint of a primitive binding, as illustrated in Figure 7.3 for two endpoints. The instantiation times for bindings with more complex configurations can be derived from the times required to create the internal features of the bindings, as seen below.

![Control interface](image)

Figure 7.3 – Test configuration for binding objects (with two endpoints)

Considering the distributed nature of the binding protocol, it is expected that the number of endpoints should not significantly impact the overall performance (as each endpoint is created in parallel by a different LBF). However, a naïve implementation has instead delivered the performance shown by the upper curve in Figure 7.4. This deviation from the expected performance was due to the direct use of the naming service to register each of the interfaces of the internal components of the binding endpoints, especially because the name server is currently implemented in a centralised way. Thus, as the number of endpoints increases, so does the number of requests to the name server, which becomes a bottleneck in the process and affects the scalability of the protocol. This problem can be solved by adopting an optimised naming service, such as with the use of replication or lazy interface registration (which is feasible as the naming scheme for the components of binding objects ensures the uniqueness of the names of their interfaces). The lower curve in Figure 7.4 illustrates the performance achieved with the second solution, when the registration of internal interfaces is deferred until binding establishment is complete. As the graph shows, binding establishment times then become nearly independent of the number of endpoints (the slight increase for each extra endpoint is due to the corresponding additional processing performed by the primary LBF, as discussed below).
In order to generalise the analysis and enable binding time estimation for arbitrary configurations, the formula below isolates the elements that compose the overall performance.

$\text{binding}_\text{inst} \equiv \text{type}_\text{access} + \text{role}_\text{iface}_\text{matchings} + \max_{i \leq \text{size}_\text{endps}} (\text{role}_\text{inst}_i) + \text{result}_\text{collation} + \text{ctrl}_\text{ifaces}_\text{creation} + \text{endp}_\text{activation}$  

(3)

The terms in the formula correspond to the steps of the protocol carried out by the primary LBF (first two and last three terms), plus the creation of the endpoints (also referred to as role instantiation) by the secondary LBFs. The amount of processing performed by the primary LBF is proportional to the number of binding endpoints (i.e., target interfaces). In particular, the matching of binding roles and target interfaces involves, in the worst case, $n_{\text{ifaces}} \times n_{\text{roles}}$ comparisons. However, as such comparisons are typically based on the repository identifier of the two interface types (unless partial matching is adopted), even the worst-case performance is not a major problem. Result collation in turn corresponds to the merging of the individual endpoint results received from the secondary LBFs, in order to produce the necessary information to activate the binding (the required time depends on the structure of the binding). Endpoint activation then basically consists in sending an asynchronous (non-blocking) message to the several involved secondary LBFs requesting the activation of each of the binding endpoints (typical times for such messages, via implicit binding, are around 2.6ms, for oneway requests of 1KB). Lastly, the creation of the binding control interfaces consists in the instantiation of the respective control components.

Finally, regarding the creation of the binding endpoints (third term in the formula), in a typical scenario, with each secondary LBF running in a different machine (and
assuming that all of them receive the request for endpoint creation at approximately the same moment), the overall delay is dominated by the time to instantiate the most complex endpoint. This is because the primary LBF remains blocked after sending all the requests for endpoint creation, until the last reply arrives (unless there is a timeout due to the failure of any of the endpoints). The time required to create each of the binding endpoints can be estimated with the formula below:

\[
\text{role\_inst} \equiv \text{type\_access} + \left( \sum_{i=1}^{n_{\text{prim\_comps}}} \text{prim\_comp\_inst}_i \right) + \left( \sum_{j=1}^{n_{\text{comps}}} \text{comp\_inst}_j \right) + \sum_{k=1}^{n_{\text{local\_binds}}} \text{local\_bind}_k + \sum_{l=1}^{n_{\text{nested\_role\_insts}}} \text{nested\_role\_inst}_l + \text{iface\_map} + \text{local\_bind} + 2 \times \text{send\_async\_req}
\]

The first four terms have similar meaning as for composite component instantiation. The fifth term, in turn, represents the time required to instantiate the role configuration corresponding to each of the nested bindings. In the case of nested composite bindings, this is a re-application of the formula. In the case of primitive bindings, however, the role is instantiated by loading and initialising the primitive binding implementation, as well as creating an interface object to represent the endpoint’s interface. The time involved in this case is thus similar to the instantiation of a primitive component with a single interface. In addition, the time for role instantiation also includes the mapping and local binding (with the target interface) of the binding interface at the particular endpoint, as well as the two asynchronous messages involved in the process (from the primary LBF to the secondary LBF, to initiate role instantiation, and on the other way round for the result).

### 7.3.3 Reflection performance

Meta-object creation and initialisation

The performance of reification is examined using several experiments that illustrate the creation of meta-objects according to the structural meta-space models described in Chapter 5. The results are discussed below (note that the implementation of the meta-space models corresponds to the default meta-objects described in Chapter 6).

1. Interface Discovery – an analysis of reification according to this meta-space model (see Chapter 5) shows that the time required to create a meta-object is independent
of the particular base-level object. In the current prototype and test environment, this time is around $8.8\text{msec}$ for components and $9.8\text{msec}$ for binding objects.

2. Interface – similarly, an analysis of the interface reification process shows that the time required to create a meta-object is independent of the particular interface definition. The average time for this process is around $9.5\text{msec}$.

3. Architecture – in this case, the performance depends on the type of the particular base-level object (more precisely, on the complexity of its configuration). Two experiments were chosen to illustrate the scalability properties of architecture reification, firstly for composite components, using linear composition and varying the number of nested components (and local bindings), and then for binding objects with an increasing number of endpoints (using the binding configurations described in 7.3.2). The results are presented, respectively, in Figure 7.5 and Figure 7.6. As the graphs show, the delay is proportional to the number of nested components, although, for bindings, it is also a function of the number of endpoints. This is because the meta-object needs to build a representation of the concrete object graph representing the configuration of the base-level object.

In the three cases, the major factor affecting reification performance is the creation and initialisation of the component corresponding to the meta-object. Currently, this is a primitive component with a single interface (although with a different definition and implementation in each case). The other factor affecting the performance corresponds to the times to check if the meta-object already exists (which is not the case) and to subsequently register it with the meta-object server ($1.9\text{ms}$ and $2.1\text{ms}$, respectively). This suggests that meta-object server optimisation can help reduce reification times.

Finally, the resulting figures indicate that meta-objects should ideally be created in advance of the need for reflection, especially in the case of Architecture meta-objects and in time-critical applications. Note that, once meta-objects are created, the time to access them is less significant (practically constant around $4\text{ms}$, when remote to the meta-object server). The results also suggest the need for more efficient meta-objects. An option is to re-implement them as separate C or C++ modules, using Python’s extension facilities [van Rossum and Drake 2001] to integrate them into the prototype.
Adaptation

Performance is a critical issue for the adaptation mechanisms, as they are meant for dynamic use, while the base-level object is operational. This is particularly an issue in applications with real-time requirements. In order to demonstrate the levels of performance in the current implementation, three experiments were carried out, using the most common adaptation operations. In order to eliminate additional overheads, primitive components with minimal implementations were used. This is because the complexity of the involved components influences the adaptation time (e.g., consider the time to instantiate a component before inserting it in a configuration). The results are shown in Table 7.2 for binding adaptation. Component adaptation, in turn, would require slightly lower times, as both base- and meta-object would be in the same capsule, eliminating the need for inter-process and network communication.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component insertion</td>
<td>37.5</td>
</tr>
<tr>
<td>Component removal</td>
<td>44.0</td>
</tr>
<tr>
<td>Component replacement</td>
<td>90.4</td>
</tr>
</tbody>
</table>
Considering the typical QoS requirements of continuous media, in particular for audio and video streams, which require delays not greater than 250ms [Hehmann et al. 1990], the above results seem appropriate. It may be feasible to schedule a given reconfiguration during the period between packet arrivals. However, especially in binding scenarios, multiple simultaneous reconfigurations are typically required, such as when inserting media filters in the several binding endpoints. In such cases, mechanisms for adaptation orchestration can be employed, in order to synchronise the reconfigurations and minimise packet losses. Such considerations reinforce the argument towards more efficient meta-object implementations, possibly in C or C++, in order to give a wider margin for the orchestration algorithms. In addition, note that the above figures should be seen as lower bounds, as the performances of instantiation and deletion of the involved components further add to the adaptation times.

### 7.3.4 Interaction performance

**Foreword**

Although the chosen implementation environment does not favour communications efficiency, this section provides an idea of the kind of performance that can be achieved with the Meta-ORB. The experiments presented here permit an analysis of the meta-model with respect to interactions, demonstrating its overhead, as well as its suitability for simple multimedia applications. Note that the figures shown here are relative to particular Meta-ORB platform configurations, instead of representing the absolute performance of the platform (as different configurations can have different levels of efficiency). In order to facilitate an analysis of the results, whenever appropriate, minimal configurations have been adopted for the experiments. In addition, in the related comparative experiments, the same level of functionality (e.g., marshalling) is present, in order to highlight the intrinsic impact of the meta-model.

**Local binding delay and throughput**

The efficiency of local bindings is an important factor in the performance of binding endpoints, as the interactions have to pass through a sequence of components locally bound via their interfaces. The other relevant factor is the efficiency of the components themselves, although this depends on their particular implementations and, thus, is not considered here (component implementations with minimal overhead
are instead employed). The experiment uses component pipelines with an increasing number of components and local bindings, as well as a request size of 1024 bytes. Note that, as Python does not support arguments by value, explicit argument copy was used in both experiments in order to force actual data transfer. However, this is as a worst-case scenario, as such data copies should normally be avoided.

Local binding delay is measured by reading the clock immediately before the first component in the pipeline issues an interaction and immediately after the last one receives it. The delay is obtained by subtracting the former reading from the latter. In addition, to show the overhead of local bindings, the figures are compared against Python method calls, using similar pipelines of objects. Figure 7.7 presents the results.

![Figure 7.7](image)

**Figure 7.7 – End-to-end delay through pipelines of primitive components and local bindings, compared with pipelines of Python method calls (using buffers of 1024 bytes)**

As the graph shows, the overhead factor is roughly around 3. This is due to the design of local bindings in the current implementation, which is derived from OOPP [Andersen et al. 2000]. Each local binding involves two levels of indirection, respectively to forward the calls to the opposite interface and for the latter to deliver the calls to its respective component implementation (thus, 3 calls, in contrast with a single one in the pure-Python experiment). In addition, the two extra calls are made through callable objects that emulate the behaviour of methods, further adding to the overhead. Although it could be avoided, such indirection will be useful in future extensions of the prototype, especially considering the support it offers for the interception of interactions, both at sender and receiver, which is crucial to an eventual implementation of the Interception meta-space model. Finally, to meaningfully interpret the figures, it is necessary to consider that local bindings are usually part of larger configurations, notably binding objects. For instance, the delays
of the local bindings in the data path along a binding object accumulate to contribute to the overall delay of the binding. Note also that the use of multiple composition levels does not affect the performance, due to the strategy used to define the interfaces of composite components and binding objects (see Chapter 4).

Note that the delay figures indicate an appropriate performance for continuous media, with a large margin considering the maximum delay requirements of audio and video interactions. However, in the particular case of real-time media, even the minimum time for traversing a pipeline of local bindings may exceed the interval between successively produced media samples (e.g., CD-quality audio presents an inter-sample interval of 23μs). This means that potential samples may be missed (i.e., not read from the media device) during the transfer of a previous sequence of samples across the local bindings. Nevertheless, this is a common problem and is usually solved with the correct management of the media device’s input buffer.

Regarding local binding throughput, due to the method used to measure the delay, estimations can be obtained by dividing the request length by the delay (in seconds). For instance, for pipelines of 3 components (2 local bindings), the maximum throughput (not considering the processing overhead of the components) is around 70Mbits/s (in comparison with 215Mbits/s for the equivalent pipeline of method calls). Note, however, that the throughput tends to rise linearly with the request size (e.g., for 32KB, the maximum throughput for the same pipeline is around 2Gbits/s).

Thus, in light of the typical throughput and delay requirements of distributed multimedia ([Hehmann et al. 1990]), as well as the maximum performances achieved in the current implementation, local bindings seem not to pose an unacceptable overhead (at least for simple media types, ranging from telephone-quality audio to compressed TV-quality video). The major performance issue is thus related to distributed bindings, which are examined next. Nonetheless, it should be stressed that the overhead generated by the particular component implementations (in a pipeline of locally bound components) should also be considered when estimating interaction performance in real applications.
Composite binding delay

The experiments below illustrate the *roundtrip delay for operational interactions*, as well as the *end-to-end delay for stream interactions*. Binding configurations similar to Figure 7.3 were used, with the aim of simplifying the analysis. In order to place the figures in perspective, a comparative analysis is drawn with the Python-based CORBA ORB Fnorb 1.1 (for operational interactions), and with TCP/IP sockets in Python.

The performance of operational interactions is exemplified in Figure 7.8 below, using a logarithmic scale. The comparison with TCP-based sockets shows an overhead of about 30% for small interactions (up to 128 bytes), although the overhead decreases for larger interactions (for interactions over 4KB, it does not exceed 10%). This extra cost can be explained as the composition of the overheads of the two local bindings at each of the binding endpoints (the time consumed by the binding components, notably for marshalling, is compensated by similar processing carried out in the sockets-based experiment). The higher level of abstraction of the Meta-ORB programming model, however, justifies such overhead. Finally, the complementary comparison shows that the Meta-ORB framework can be used to achieve superior performance in comparison with Fnorb (which is at least about 5ms slower in all cases). Note that the binding used in the experiment, as well as the program used in the sockets-based experiment, mimic the externally visible behaviour of Fnorb (although in an explicit binding context)\(^3\).

![Figure 7.8 – Delay for synchronous requests through a simple distributed binding](image)

\(^3\) There are, however, differences in the internal behaviour, notably related to sophisticated buffer management and message segmentation algorithms employed in Fnorb, which do not have a counterpart in the platform configuration used in the experiment (neither in the sockets experiment).
Chapter 7 – Evaluation

An analysis of the end-to-end delay of stream interactions, in turn, comparing a Meta-ORB binding with UDP sockets, is shown in Figure 7.9. As it is not physically possible to measure the actual end-to-end delay between independent network nodes, an approximation method was instead adopted. In particular, both experiments estimate the delay by making the stream consumer echo each frame back to the producer and then dividing the corresponding roundtrip delay by two. Although this method is not accurate, as it does not consider the influence of the operating system scheduler (which may delay the consumer thread that sends the echo message back), the precision obtained is sufficient for the present analysis.

![Figure 7.9 – Estimation of the end-to-end delay for stream interactions](image)

Compared with the typical delay requirements of multimedia applications, again referring to the results presented in [Hennemann et al. 1990], the figures seem appropriate (although the achievable throughput is another constraint, as seen below). However, the extra overhead required for multimedia processing must be accounted for, suggesting the implementation of computation-intensive components in C or C++, and then using Python as the glue for their integration into the platform (see further discussion in section 7.3.5). In practice, thus, the above results should be seen as lower bounds for the delay, with any extra overhead at the endpoints (e.g., resulting from additional local bindings and stream processing) adding up to the figures.4

4 In both experiments the only significant extra overhead at the endpoints results from marshalling.
Composite binding throughput

While throughput can in principle be computed from the delay figures shown above, the actual method used for measuring the delay, as already discussed, would not enable accurate values. Thus, a dedicated experiment was designed to measure the maximum throughput of stream bindings, and also to show the overhead in comparison with UDP sockets. The method used involves timing the reception of the first and last packets in a sequence and then dividing the total amount of data received (in megabits) by the elapsed time. Note that, although this method does not consider the effects of buffering performed by the underlying protocols (such as an initial delay), it provides a view of the throughput as perceived by the stream consumer. In addition, note that the experiments were carried out with only one station (the one where the stream producer was located) transmitting significantly on the Ethernet. While this suffices for estimating the cost of the programming model, in real scenarios the impact of network contention should be taken into account (although this is typically a concern for QoS management mechanisms, which can be implemented on top of the platform). The experimental results are presented in Figure 7.10.

![Figure 7.10 – Throughput of stream interactions, compared with UDP sockets](image)

As the graph shows, the absolute overhead in relation to UDP sockets is roughly the same for frame sizes over 32 bytes (in practice, the difference is in between 300Kbits/s and 700Kbits/s). In relative terms, the overhead is around 60% for frame sizes up to 32 bytes, decreasing to less than 10% for frames over 512 bytes. As in the delay analysis, the main cause of overhead is the extra time required for interactions to traverse the local bindings in the data path. From the figures, however, it can be seen that the use of larger frame sizes helps to compensate such overhead. Note, though,
that the extra time consumed by the media processing components may also affect the throughput, meaning that the above figures represent the maximum achievable values (when the only processing overhead refers to marshalling). Nevertheless, the same considerations apply to a sockets-based program exhibiting similar media processing functionality.

Finally, to put the figures in perspective, they can also be compared with typical requirements of continuous media. In particular, for audio streams, the performance is more than enough. In fact, an experiment used to read audio data from a file and distribute it to a number of consumers (using 8 bits sample size and a play-out rate of 11025 samples per second, with 1024-byte frames) has shown that a delay of about 10ms was necessary at the producer side in order to avoid swamping the consumers with samples. For video streams, however, the achieved performance is less than adequate. Nevertheless, considering the requirements of compressed TV-quality video, which range from 2-10Mbit/s, as well as the use of large buffer sizes, reasonable results may be achieved. Such an experiment, however, is an issue for future work.

### 7.3.5 Discussion

This section has presented an analysis of the quantifiable aspects of the Meta-ORB architecture. The study of the establishment mechanisms has demonstrated the performance for instantiating the several kinds of meta-model constructs. In addition, the study has provided a method to estimate the instantiation time for arbitrary configurations. This analysis, however, was not an end in itself. Instead, its main aim was to provide support to understand the performance of the reflection mechanisms\(^5\). Firstly, component instantiation plays an important part in the creation of meta-objects, suggesting that simplified meta-object configurations usually help reduce reification time. Secondly, the performance of component instantiation is also an important aspect of adaptation. As a general rule, adaptation time grows with the complexity of the handled components, due to the time for their instantiation and/ or deletion. This also suggests that the implementation of component factories is critical for adaptation efficiency and should be optimised (possibly implemented in C/ C++).

---

\(^5\) Although, in real-time applications, establishment performance can be a major factor, as there may be requirements regarding the set up time of interaction sessions and their respective binding objects.
Regarding interaction performance, the analysis has considered the overhead of the meta-model in comparison with a sockets-oriented baseline. The experimental results suggest that the overhead is compatible with the higher abstraction level of the meta-model. In addition, the comparison with Fnorb demonstrates the benefit of an open architecture, in which the platform can be properly configured to yield better performance. In general terms, this suggests that the overhead incurred by the use of an explicit meta-model for building middleware can be compensated by exploiting such openness to introduce application-specific optimisations.

More specifically, the analysis has concentrated on two variables, delay and throughput, and, similarly to the establishment mechanisms, a study of the performance of primitive interactions was provided, in order to enable the estimation of the interaction performance of arbitrary platform configurations. Note that while other important variables could have also been studied, such as delay jitter and error rates, especially under realistic network loads, such a study would only make sense if the appropriate QoS management mechanisms were in place. Thus, a study of these performance variables is considered out of the scope of this thesis. Nonetheless, the provided analysis permits us to conclude that the meta-model, even in a non-performance-based implementation, satisfies the basic requirements of simple multimedia applications. In addition, the analysis has suggested that an effective way to improve interaction performance is to implement computation-intensive components, as well as primitive bindings, in an efficient compiled language (C or C++). Such components can then be seamlessly integrated into the current prototype, using Python’s ability to be extended with external C/C++ modules [van Rossun and Drake 2001]. In this case, the major persisting overhead (in comparison with a pure C/C++ implementation) would be associated with local bindings. Nevertheless, as demonstrated, local bindings permit reasonable performance, while also contributing to a higher-level programming model (although better efficiency would still be feasible by avoiding the extra levels of indirection involved in a local binding).
7.4 Qualitative evaluation

7.4.1 Overview

While the quantitative analysis enables the evaluation of important properties of the Meta-ORB architecture, other essential aspects of the proposed programming model are not easily or meaningfully quantifiable. In particular, the programming model must be evaluated from the point of view of its usability, in order to demonstrate how the goals of configurability and reconfigurability can be achieved. Hence, in this section, the major aspects involved in accomplishing such goals are studied from a qualitative point of view, in an effort to highlight the more abstract properties of the architecture. The analysis is based on an experimental application scenario, described in 7.4.2, which has been developed to demonstrate the main features of the platform. The use of the meta-model to configure this scenario is discussed in 7.4.3. Dynamic reconfiguration is then discussed in 7.4.4, demonstrating the use of the MOP to adapt the structure of the platform. Subsection 7.4.5 considers the type evolution support, based on the use of reflection and type management to dynamically generate and reuse configuration descriptions. Finally, subsection 7.4.6 discusses the role of meta-information management regarding the above three aspects.

7.4.2 The case study: requirements and architecture

The application scenario used in the analysis is depicted in Figure 7.11, and consists of an interactive audio distribution application, supported by a multi-point binding with stream interfaces. The audio producer is a component that periodically reads a number of samples from the local audio device and sends them through its interface, using an ‘out’ flow. The consumer components, in turn, receive such audio samples through an ‘in’ flow in their interfaces and immediately write such samples to their respective audio devices for play-out. The application uses PCM-encoded CD-quality audio (giving a throughput requirement of 1.41Mbit/s). By default, audio compression is not required, in order to save processor cycles. However, if the available network bandwidth drops, compression must be introduced to lower the throughput requirements. The supporting binding is defined with two roles, corresponding to the producer and consumer sides of the application. Each role defines a minimal endpoint configuration, consisting of a stub and a primitive binding
endpoint. The primitive binding is based on a simple encapsulation of the UDP protocol (although the use of a multicast protocol would have been more appropriate in this particular case). In addition, the binding has a control interface, with operations to allow the audio flow to be paused and re-started (which in turn is done via implicit binding, using the control interface of the component representing the producer stub).

Figure 7.11 – Binding configuration for the audio distribution case study

The complete type definitions and Python code for the several elements used in this case study can be found in the Meta-ORB web site [Costa 2001] (although the current implementation does not include the facilities for network monitoring, meaning that external mechanisms must be used to direct the adaptation functions).

7.4.3 Configuration

Object definition

The definition of the binding type (as introduced above) is presented in Figure 7.12, along with the primitive binding type used as part of its configuration (for brevity, the definition of the internal components and interfaces are not shown). This exercise shows that most of the features of a binding definition are specified in terms of the binding roles, including the declaration of target interfaces and the description of the corresponding endpoint configurations. This endpoint-based style of binding definition has greatly facilitated the development of the distributed binding protocol, as it enables the prompt identification of the portion of the binding configuration that must be attributed to each local binding factory as part of the binding establishment process. In addition, the approach permits complete flexibility as to which components should be part of an endpoint configuration, as well as regarding their interconnection.
module CaseStudy {
    primitive binding PrimBindingUDP {
        implementation: PrimBindingUDP_impl;
        role Sender {
            target interface: SenderInterf;
            cardinality: 1;
        };
        role Receiver {
            target interface: ReceiverInterf;
            cardinality: 1..5;
        };
    };
    binding AudioBinding {
        control interfaces: BindCtrlInterf ctrl is (CtrlComp, ctrl_interf);
        internal bindings: PrimBindingUDP prim_binding;
        role Producer {
            components: ProducerStub prod_stub;
            target interface: ProdInterf is (prod_stub, consumer_interf);
            cardinality: 1;
            configuration: (prod_stub, sender_interf):(prim_binding, Receiver);
        };
        role Consumer {
            components: ConsumerStub cons_stub;
            target interface: ConsInterf is (cons_stub, producer_interf);
            cardinality: 1..5;
            configuration: (cons_stub, receiver_interf):(prim_binding, Sender);
        };
    };
}

Figure 7.12 – Definition of the composite and primitive bindings used in the case study

However, as can be seen from the definition of target interfaces, binding definitions currently depend on the particular interfaces to be bound, thus affecting the potential for reuse of existing binding types. In the worst case, it is as if the applications programmer had to also build the middleware as part of application development. This clearly is a problem and needs to be avoided. However, it must be remembered that this is a direct consequence of the degree of configurability of the architecture. This is in contrast with conventional middleware, where the platform internals are transparent to the programmers, although at the price of a rigid and non-configurable service. Thus, considering the need for flexibility imposed by dynamic applications, as well as the improvements discussed below, the extra cost of the approach can be justified.

Improvements

The above problem can be minimised by defining standard interfaces for components (and also bindings) that are intended for use as part of binding configurations. One option that has been adopted so far exploits the fact that the complex interaction styles (operational and stream) can be decomposed in terms of simpler signal interactions (see Chapter 4). In the example above, this is represented
by the use of sender/receiver interfaces, which define valued signals to convey interactions in a generic way. Importantly, the types for such signals (and their values) should ideally be standardised, in order to improve interoperability. In this respect, a possibility that has been investigated (though not fully implemented) consists in using the General Inter-ORB Protocol (GIOP) [OMG 2001a] for encoding operational and signal interactions (with an extended message type for the latter), as well as the Simple Flow Protocol (SFP) [OMG 2000a], for stream interactions. Crucially, the overall approach makes possible the provision of a library of standard binding and component types, which the developers can retrieve from the Type Repository and use when assembling application-dependent binding configurations. An example is the above primitive binding definition, which is generic and can be used across a number of binding configurations with similar requirements. Note that further improvements leading to greater reusability are possible, as will be considered in Chapter 8.

Instantiating the application scenario

Typically, a scenario such as the one described above requires third-party binding. Thus, a component in charge of application management (or even the user him/herself) would be responsible for identifying the interfaces of the components to be bound, as well as the binding type, and then requesting the local binding factory to create the binding. Assuming that the application components have already been created, this would be done with the following code, adapted from the implementation of the case study (and considering the use of the default binding factory described in Chapter 6):

```python
# Import and initialise the platform modules.
import MetaORB

# Obtain an interface reference to the local binding factory.
BF = MetaORB.capsule_mngr.BF

# Assemble the list of target interface references (for brevity, the
# process of obtaining the irefs has been omitted from this example).
target_iref_list = [iref_prod, iref_cons1, iref_cons2, iref_cons3]

# Request binding establishment, getting an IRef to the control
# interface of the new binding.
bind_ctrl = BF.new(target_iref_list,
                   'M-ORB:comp.lancs.ac.uk/CaseStudy/AudioBinding:1.0',
                   'Binding001')
```

6 The adoption of these standard OMG protocols facilitates an eventual effort towards interoperability with CORBA (although full conformance with other aspects of these protocols would be necessary).

7 This approach, however, does not apply to top-level binding types, which are application-specific.
Note that one way to simplify the above request would be to have the binding type specified as part of the target interface references. This would however introduce a further criterion for interface compatibility, although multiple binding types could be specified and binding type negotiation supported. In addition, this would require the corresponding changes to binding factories and to the structure of interface references.

Discussion

In comparison with CORBA and other conventional middleware platforms (such as those examined in Chapter 2), the Meta-ORB programming model introduces a further level of flexibility. While conventional platforms have fixed, typically hard-coded configurations, the Meta-ORB is completely flexible, allowing the developer to freely configure (and dynamically reconfigure) the platform. Although this flexibility introduces the complexities discussed above, it is clearly needed for a category of applications that are not properly supported by conventional middleware. This includes distributed multimedia applications, as well as those involving mobility, real-time, and context-awareness. In common, such applications require control over the underlying middleware support, notably for its customisation. This requirement is fulfilled, in the Meta-ORB, with the use of a uniform and explicit meta-model, which applies to both platform and application configurations.

On the other hand, for less demanding categories of application (such as in conventional client/server scenarios), the kind of middleware support offered by mainstream middleware technologies continues to be appropriate. This means that, in order to support such applications in the Meta-ORB, it would be more appropriate to use implicit bindings (instead of having to define explicit, composite bindings).

As a last remark, the backward compatibility with the CORBA meta-model, as discussed in Chapter 4, means that CORBA IDL definitions can be reused in the context of the Meta-ORB. This also contributes to an eventual effort towards interoperability with the OMG standard, since the same set of basic types is adopted. However, it must be noticed that some constructs inherited from CORBA, particularly interfaces, although preserving the original semantics, may have different models of use in the Meta-ORB. This is due to the first-class status of interfaces, and also to the explicit association of interfaces to components and to the use of explicit binding.
7.4.4 Adaptation

Foreword

Static configuration, as described above, is an important requirement of emerging application areas. However, because of their dynamic behaviour, support for runtime reconfiguration must also be provided. In the Meta-ORB, such support is achieved through reflection and meta-object protocols. This is facilitated since all aspects of a platform configuration are described and handled in terms of a uniform meta-model, which also provides the basis for reification. In what follows, dynamic reconfiguration of the platform is demonstrated, based on the case study described above. The analysis will consider the reconfiguration options, their programmability and the need for future enhancements. It must be emphasised that the goal of the current MOP is to offer the basic support for adaptation instead of providing for its management. In particular, the latter involves the definition of when and how reconfigurations should be carried out. This is currently a task for the users of the MOP.

Programming adaptation

The need for adaptation in the case study originates from the requirement to match the currently available network bandwidth with the throughput requirements of the binding configuration, considering a trade-off between the use of bandwidth and the processing resources at the endpoints. This is achieved by inserting compressor/ decompressor components when the bandwidth becomes scarce and removing them when the bandwidth recovers. The analysis assumes that the configuration is up and running, and that the types of the compressor and de-compressor components have already been defined and are compatible with the interfaces of the other binding components. In addition, the analysis assumes that an appropriate adaptation manager is in place to monitor the throughput and direct the use of the Architecture MOP for the necessary reconfigurations.

The basic pre-requisite for adaptation is to obtain an interface reference to the Architecture meta-object that reifies the particular binding object (using the unique name of the binding, as obtained from the interface reference that was returned by the binding factory, as seen in 7.4.3):
import MetaORB

# Use the basic MOP to create/retrieve the Architecture meta-object.
arch_mobj = MetaORB.get_arch_mobj(bind_ctrl.get_binding_name())

Subsequently, the adaptation manager must obtain meta-information describing the
binding configuration, so that it can identify the points where to perform the necessary
reconfigurations. This can be done via introspection, by querying the meta-object for
the concrete object graph corresponding to the binding:

obj_graph = arch_mobj.get_obj_graph()

The contents of the returned data structure should look like in the table below (for a
binding with three endpoints), where each row represents a local binding in the graph,
in terms of the unique names of the involved components and their bound interfaces.

<table>
<thead>
<tr>
<th>Component uname</th>
<th>Interface uname</th>
<th>Comp. (binding) uname</th>
<th>Interf. (endpoint) uname</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod_stub:Producer@capsule001:Binding001</td>
<td>sender_interf:prod_stub:Producer@capsule001:Binding001</td>
<td>prim_binding:Binding001</td>
<td>Receiver@capsule001:prim_binding:Binding001</td>
</tr>
<tr>
<td>cons_stub:Consumer@capsule002:Binding001</td>
<td>receiver_interf:cons_stub:Consumer@capsule002:Binding001</td>
<td>prim_binding:Binding001</td>
<td>Sender@capsule002:prim_binding:Binding001</td>
</tr>
<tr>
<td>cons_stub:Consumer@capsule003:Binding001</td>
<td>receiver_interf:cons_stub:Consumer@capsule003:Binding001</td>
<td>prim_binding:Binding001</td>
<td>Sender@capsule003:prim_binding:Binding001</td>
</tr>
</tbody>
</table>

Given the above meta-information, the adaptation manager will be able to uniquely
identify the points where compressor (and de-compressor) components should be
inserted. In particular, the unique names of interfaces and nested binding endpoints
(which act like interfaces) are used to mark the local bindings that will be replaced by
the new components. Thus, when the need for adaptation is detected (i.e., a drop in the
available bandwidth), the adaptation manager invokes the meta-object to perform the
necessary reconfigurations, such as in the following code (extracted and adapted from
the case study implementation):

# Insert the compressor filter in the producer endpoint.
position = MetaORB.InsertLocation(
    'sender_interf:prod_stub:Producer@capsule001:Binding001',
    'Receiver@capsule001:prim Binding001')
arch_mobj.insert_component(
    'M-ORB:comp.lancs.ac.uk/CaseStudy/CompressorComp:1.0',
    'compressor', position)
# Insert the de-compressor filters at the consumer endpoints. These # two statements would have to be properly repeated for each endpoint # conforming to the Consumer role.
position = MetaORB.InsertLocation(
    'receiver_interf:cons_stub:Consumer@capsule002:Binding001',
    'Sender@capsule002:prim_binding:Binding001')
arch_mobj.insert_component(
    'M-ORB:comp.lancs.ac.uk/CaseStudy/DecompressorComp:1.0',
    'decompressor', position)

Conversely, when the adaptation manager detects a consistent rise in the available bandwidth, it restores the original binding (by removing the filters), in order to make the binding endpoints more lightweight again. This would be done as shown next (note that the unique names of the components to be removed would have been obtained using similar introspection as described above):

# Remove the compressor filter from the producer endpoint.
arch_mobj.remove_component(
    'compressor:Producer@capsule001:Binding001')

# Remove the de-compressor filters from the consumer endpoints. As # above, this statement would have to be properly repeated for # each endpoint conforming to the Consumer role.
arch_mobj.remove_component(
    'decompressor:Consumer@capsule002:Binding001')

From the above examples, it can be seen that the operations used to insert and remove components offer a level of transparency over the underlying mechanisms of component creation, deletion and local binding. In addition, location transparency is promoted, as the same operations permit the actuation on local and remote endpoints alike (the fact that the location of the handled entities is apparent from the unique names used is a mere consequence of the particular naming scheme adopted in the prototype). Nevertheless, it can also be argued that the degree of repetition is considerable, as the insert and remove operations need to be explicitly invoked to reconfigure each of the endpoints conforming to the consumer role. In order to solve this problem, the Architecture MOP also provides role-based operations, which require a single step to perform a uniform reconfiguration on all endpoints conforming to a role. In what follows, it is shown how the reconfiguration to insert the filters was re-implemented using role-based component insertion. It must be noticed, however, that conventional, non role-based reconfiguration is still useful (and indeed the only option) to adapt composite components and individual binding endpoints (in case the adaptation condition holds for one endpoint but not for the others of the same role).
# Insert the compressor component in the Producer role.
node1 = MetaORB.GraphNode('prod_stub', 'sender_interf')
node2 = MetaORB.GraphNode('prim_binding', 'Receiver')
position = MetaORB.LocalBindingDcl(node1, node2)
arch_mobj.role_insert_component('Producer',
    'M-ORB:comp.lancs.ac.uk/CaseStudy/CompressorComp:1.0',
    'compressor', position)

# Insert the de-compressor component in the Consumer role.
node1 = MetaORB.GraphNode('cons_stub', 'receiver_interf')
node2 = MetaORB.GraphNode('prim_binding', 'Sender')
position = MetaORB.LocalBindingDcl(node1, node2)
arch_mobj.role_insert_component('Consumer',
    'M-ORB:comp.lancs.ac.uk/CaseStudy/DecompressorComp:1.0',
    'decompressor', position)

Note that the role-based operations usually involve a slightly more complex list of arguments, especially related to the different way of specifying the point where the reconfiguration should take place. In the above example this corresponds to the use of a data structure representing the (abstract) local binding that will be replaced by the inserted component. Specifying the position of reconfigurations in this way is necessary since unique interface names do not apply to role configurations (which are abstract). In practice, though, such data structures do not need to be explicitly assembled, as they can be retrieved from the meta-object using role-based introspection (more precisely, using the get_role_config operation of the Architecture MOP). Note that this discussion does not apply to role-based component removal, which only requires the (non-qualified) name of the component to be removed and the name of the involved role.

Importantly, as discussed in Chapter 5, besides enabling more powerful operations, the role-based approach also implies the evolution of the type of the base-level object. This means that the original type no longer serves as a correct description the object. Thus, all subsequent meta-information about the object should be obtained from the meta-object, using introspection. Note that a similar effect is caused by the expose_interf operation, which enables new interfaces to be added to composite components. Such issues will be further considered in subsection 7.4.5.
**Discussion**

The above experiments have demonstrated the use of the default MOP to meet the runtime adaptation requirements of dynamic applications. In particular, the examples have shown how configurations can be adapted, by dynamically adding and removing components. More generally, as the same constructs used to build platform configurations can be manipulated by meta-objects, all aspects of the structure of a platform can be subject to adaptation. Thus, the same power of expression of the configurability mechanisms is also available for reconfiguration.

While other reflection operations (see Appendix C for a complete list) have not been examined, similar considerations generally apply to them as well. In addition, regarding introspection, particularly via the Interface and Interface Discovery metaspace models, the use of the corresponding facilities in the MOP is typically straightforward, as they basically provide access to type and other runtime meta-information. Importantly, due to the possible effects of type evolution, the use of such facilities should be preferred for objects that have been subject to reconfiguration (instead of directly accessing the types of such objects in the repository).

Note that, although the case study represents a category of applications where adaptation is performed in an automatic way (through an adaptation manager), the model is also useful for interactive adaptations. This is particularly the case with long-lived, non-stop applications for which the requirements may change in ways that cannot be predicted beforehand. In such cases, human intervention may be required in order to decide the kind of reconfigurations that are necessary. Currently, this use of the MOP is realised through the interactive Python prompt. Note, however, that interactive adaptations would greatly benefit from the existence of a friendlier user interface (such as a console-like GUI), which would enable the visualisation of configurations and the expression of reconfiguration actions in a more intuitive way. This is an issue for future work and will be considered in Chapter 8.

Other important aspects of the MOP, especially regarding reconfiguration, are related to the issues of safety and security. The former refers to the need to ensure the integrity of a binding configuration after adaptation takes place, while the latter is important to prevent malicious or unauthorised use of reconfiguration, as this can affect the proper functioning of the platform. Currently, though, only elementary treatment is provided, regarding the issue of security. This is based on the requirement
for meta-level clients to supply the unique identifier of an object before access to the corresponding meta-object can be granted. Clearly, future work in this area is needed, as will also be considered in Chapter 8.

Finally, the above analysis should be placed in perspective, considering that the Meta-ORB architecture does not prescribe a fixed meta-object protocol. In fact, as discussed in Chapter 6, the meta-level can be extended with new meta-object types and their respective MOPs. Thus, it is feasible to implement enhanced reflection capabilities as meta-level extensions, which would be able to co-exist with the default MOP. For instance, it would be possible to have a specialised MOP for use with safety-critical applications, as well as a simpler, lightweight MOP for applications without such constraints. Note that all possible MOPs must be defined and implemented in terms of the constructs of the meta-model. In particular, the finest granularity level at which functionality can be dynamically introduced in a configuration corresponds to primitive components. This is because changes at a lower level (e.g., adding methods to an interface or adding interfaces to primitive components) would require the use of language-specific features (such as the dynamic downloading of methods), thus contradicting the basic principle of language independence, which is crucial for middleware.

7.4.5 Type evolution

Motivation

In the above case study, assuming that the low-bandwidth situation becomes common, it may be more efficient for applications to start up using the binding configuration that includes audio compressing. In such cases, instead of statically defining a whole new binding type for this purpose, the results of previous adaptations can be used. This is enabled by the type evolution feature, which combines reflective adaptation with meta-information management (represented by the Type Repository). In the following, the several issues involved in the dynamic definition and use of new type definitions are discussed, along with the potential applications of the concept. The application scenario presented in 7.4.2 is used as the running example.
Developing and reusing dynamically generated configurations

As mentioned above, a crucial consequence of role-based reconfiguration is the change of the type of the base-level object, whereby the new effective type is represented by the meta-information maintained in the meta-object. As seen in Chapter 5, though, the new type cannot be used for other purposes unless it is first published, which can be achieved with the following simple statement:

new_rep_id = arch_mobj.commit_type()

This causes the creation of a new type in the repository, with a repository identifier equal to (in the example): “M-ORB:comp.lancs.ac.uk/CaseStudy/AudioBinding:1.1”. Note the version number, which distinguishes the new type from the original one.

Similar considerations apply to the evolution of component types, although, in this case, the sources of type evolution are different: the type can evolve either by adding a new interface to a component or by explicitly deriving a new type from an adapted configuration. In both cases, though, the same commit_type operation is used.

Importantly, once published, new type versions can be used in the same way as statically defined types. In the example, a binding object of the new type could be created by invoking the binding factory in the normal way, such as in:

bind_ctrl = BF.new(target_iref_list,
   ‘M-ORB:comp.lancs.ac.uk/CaseStudy/AudioBinding:1.1’,
   ‘Binding002’)

Discussion

As seen from the example, support for type evolution is provided in a very easy-to-use way. The main complexity is related to the actual reconfigurations that cause type evolution in the first place (although such complexity is not directly related to type evolution, being an inherent issue in environments that offer dynamic reconfigurability in such a flexible way). Crucially, though, the type evolution support enables the efforts spent in reconfiguration to be reused in different instances, thus spreading the associated costs (especially if human intervention is involved) and reducing the need to explicitly develop new configurations. In particular, it is considered that, once a library with representative configuration types is in place, a substantial part of the development of new configurations will be via type evolution.
Interestingly, this also suggests the use of reflection and type management to support an evolution-based style of development. As a major consequence, besides those discussed above, it would be possible to test the adapted configurations at the same time as changes are introduced. Then, as such configurations become stable, the corresponding type descriptions can be published in the repository. Although such a style would, in most cases, not account for the whole development process, it would certainly be helpful when tuning the properties of the configurations being developed.

Finally, note that although major emphasis is given to the reuse of adapted configurations, the support for type evolution has other applications. In particular, reconfigurations involving the addition of new interfaces to a composite component can be used to introduce new kinds of services, which can be reused by publishing the respective component types. In this respect, a promising approach would be to introduce a new meta-type to describe the properties of services (i.e., service types), to which the same principles of type evolution could be applied.

7.4.6 Overall remark: the central role of meta-information management

In common to all aspects examined in this section, meta-information management, represented by the meta-model and the Type Repository, must be recognised as a central enabling concept. The extensive use of the technique permits the unification of the different mechanisms defined in the architecture, namely configuration, dynamic reconfiguration and type evolution. This means that the same meta-model constructs are used during all phases of a platform or application lifecycle (from design to runtime). In addition, meta-information describing such constructs is available in the Type Repository during all of these phases (as well as in the meta-objects at runtime). This greatly contributes towards a seamless and more principled design, also effectively reducing the time necessary to master the different aspects of the architecture.

7.5 Summary

This chapter has presented an evaluation of the Meta-ORB architecture and its current implementation. The adopted approach divided the evaluation into two parts. Firstly, the quantitative evaluation examined the performance-related aspects of the architecture, demonstrating the overhead of the meta-model, as well as providing a
preliminary study of its suitability for distributed multimedia (based on the prototype described in Chapter 6). Secondly, the qualitative evaluation analysed the more abstract aspects, notably the programming model and its support for configuration, dynamic reconfiguration and type evolution. In addition, in both cases, potential enhancements were identified and discussed.

The quantitative analysis has demonstrated the suitability of the meta-model for multimedia applications. This has been verified, in absolute terms, by the current prototype and its use for simple applications involving continuous media. Most importantly, however, was the comparative analysis with sockets in Python, which showed that the overhead of the meta-model (for interactions), besides being at a reasonable level, also scales well. This suggests that its implementation in a more efficient language would have the same relative properties, thus offering suitable levels of performance for more demanding applications.\(^8\) In addition, the study of the establishment mechanisms showed that the performances are acceptable for use in dynamic reconfiguration, although further optimisations are clearly necessary.

To complement this, the qualitative analysis has demonstrated that the overall goals of the work have been achieved. The analysis has verified the flexibility of the architecture, which enables the customisation of the platform for particular applications, also allowing dynamic modifications. The study has also enabled us to demonstrate the efficacy and the usability of the proposed architecture in a realistic application scenario. These aspects are facilitated by the use of a uniform meta-model throughout the lifecycle of a platform and across the several elements that constitute its configuration. The approach enables all features of a platform configuration to be handled using a relatively small set of constructs and meta-level operations. In addition, the overall effectiveness and usability of the approach is further enhanced by the novel introduction of type evolution in combination with reflection, which permits the results of adaptations to be reproduced in a straightforward and effective way.

\(^8\) This conclusion is based on the assumption that the intrinsic overhead of the Python interpreter is not significantly greater than the overhead specifically generated by the implementations of sockets and Meta-ORB bindings. In this case, the differences measured in the experiment (such as in Figure 7.10) can be roughly seen as proportionally invariable, irrespective of the implementation language.
Chapter 8  Conclusions

8.1 Introduction

This thesis has investigated the use of meta-level techniques as a principle for the design and implementation of open middleware platforms. In particular, meta-information management has been presented as a central concept for structuring middleware, enabling an explicit treatment to the constructs used in platform configuration. Object-oriented reflection, in turn, has been adopted as the foundation for the dynamic inspection and evolution of platform configurations. Crucially, the use of the two techniques has been proposed in an integrated way, contributing to the uniformity of the mechanisms used for static configuration and dynamic reconfiguration. The main motivation for the approach has been the need to properly support applications with dynamic requirements, which demand flexible and adaptable middleware support, in a way that is not provided by current mainstream technologies.

The chapter is structured as follows. Section 8.2 revisits the main issues considered in the thesis, along with the major arguments, on a chapter-by-chapter basis. Section 8.3 then discusses the main contributions of the thesis, as well as other important results. A further assessment of the proposed approach is presented in section 8.4, by verifying the fulfilment of the reflective middleware requirements introduced in Chapter 2. This is followed by section 8.5, which discusses the main areas for future work. Finally, section 8.6 concludes the thesis with some summarising remarks.

8.2 Thesis overview

The overall context and motivation for the research were introduced in Chapter 1, which considered current middleware technologies and their lack of flexibility to support important new areas of application. The chapter introduced the two major techniques used in this thesis, namely reflection and meta-information management, also considering their complementary nature. It was argued that the combination of these two techniques represents a promising approach to achieve the required flexibility in a principled and comprehensive way. The remainder of the chapter then considered the main aims and the overall structure of the thesis.
Chapter 2 presented the first part of the research survey, which considered the principles of reflective architectures and their major areas of application, notably middleware. Crucially, this preliminary study enabled the identification of a set of requirements and principles that are considered important for reflective middleware architectures. The chapter then discussed the trend towards the use of reflection-like facilities in mainstream middleware technologies, although such use is typically *ad hoc* and lacks an appropriate level of flexibility. Important related work in the area of reflective middleware was also discussed, concentrating on the use of reflection at a more fundamental level, and comparing such work with the reflective middleware requirements. In particular, this showed that, although significant advances have been made, important aspects are still unfulfilled. Finally, the chapter considered other research areas that offer complementary techniques to reflection.

Chapter 3 concluded the survey with a study of meta-information management, again considering the fundamental principles and their application in middleware. Importantly, it was argued that although the technique enables an appropriate approach for middleware configuration management, it is not suitable for runtime adaptation, due to its intensional nature. This observation motivated the identification of a crucial relationship with reflection, providing the basis for the approach proposed in the thesis. The chapter also considered relevant standards and technologies for the management of meta-information, with emphasis on the ones used in the thesis.

Chapter 4 then explored the use of meta-information management to develop the basic aspects of the middleware architecture proposed in this thesis, called Meta-ORB. The architectural building blocks are presented in terms of a meta-model, which enables their management as first-class meta-information entities. In this respect, the meta-model also defines the structure of the meta-information management facility that is associated with the architecture. Crucially, the chapter advocates the use of this meta-model as the way to express and handle all other aspects of the platform, notably its configuration facilities and its reflective meta-level. This is possible since the same meta-model constructs are used to describe the platform at design and deployment time, as well as at runtime. In addition, the meta-model encourages modularity and reuse of platform configurations, due to its component-based nature.

The other main aspect of the architecture, its reflection framework, was presented in Chapter 5. The chapter started with a discussion of the fundamental concepts of the
meta-level, which are derived from the Open-ORB reflective middleware architecture developed at Lancaster. Notably, the use of a multi-model reflection framework was adopted as the basis for the organisation of the meta-level, in order to reduce its complexity. The chapter then presented the novel features of the Meta-ORB architecture, based on an approach to integrate meta-information management and reflection. In particular, the definition of meta-objects and reification actions in terms of the meta-model was proposed, followed by an approach to dynamically generate, via type evolution, meta-information describing new platform configurations.

Complementing the abstract description of the architecture presented in Chapters 4 and 5, Chapter 6 presented the concrete design and implementation of a proof-of-concept prototype. The description considered the three major aspects of the design. Firstly, the runtime representation of the major meta-model constructs was presented, as well as the mechanisms for their instantiation. Following this, the implementation of the Type Repository was described, which conforms to the structure of the meta-model. Finally, a realisation of the meta-level was proposed, based on a default meta-object protocol, which can be extended in future implementations. Importantly, the chapter also highlighted the nature of the Meta-ORB architecture, which consists of a framework for the definition of customised and adaptable platform configurations, regarding both the base- and meta-level functionality.

An evaluation of the proposed approach was presented in Chapter 7, considering the quantitative and qualitative aspects of the architecture. The quantitative analysis mainly focused on demonstrating the relative properties of the meta-model, notably overhead and scalability. As a main outcome, it was concluded that the extra cost is acceptable, in light of the flexibility that the meta-model enables. The analysis was complemented by a preliminary evaluation of absolute performance, showing the suitability of the current prototype for simple applications involving continuous media. The qualitative evaluation, in turn, concentrated on demonstrating the usability and effectiveness of the architecture and its programming model in a representative application scenario, considering the overall goals of configurability and reconfigurability. The study enabled conclusions about the benefits of the approach, notably the uniformity of the programming model and the positive impact on the development of customised platform configurations.
8.3 Results

8.3.1 Main contributions

Reflective middleware architecture

This thesis has proposed an architecture for reflective middleware, called Meta-ORB. The architecture allows the development of customised platform configurations, also providing for their runtime inspection and adaptation. Importantly, the architecture has been defined in terms of a framework for the base- and meta-level aspects of reflective middleware, allowing both aspects to be specialised and extended for particular application areas. This represents a novel and generic approach for flexible middleware platforms, which enables middleware support to be tailored to meet the specific and dynamic requirements of applications. In addition, a concrete design and implementation of the framework has been developed, demonstrating the feasibility of the approach. The fundamental concepts used in this approach correspond to the other major contributions of this thesis, as discussed below.

Meta-information management approach for middleware configurability

This thesis has considered platform configurability as one of the main requirements of important emerging applications of middleware. To support this, the thesis has proposed the first-class treatment of the constructs used to build middleware configurations. This is based on an explicit meta-model, with meta-types that support the definition and hierarchical composition of the individual elements of a platform, in terms of components and explicit binding objects. The approach represents a generic solution to the lack of flexibility that characterises current middleware technologies. In particular, such an approach enables a uniform programming model for developing middleware, facilitating the definition, dissemination and reuse of customised platform configurations, which can be optimised for specific application requirements. In addition, the use of meta-information management techniques, based on the MOF, enables the extensibility of the meta-model, which can hence be augmented with new constructs. Such techniques also facilitate the definition and evaluation of relationships between meta-information elements (e.g., type substitutability and compatibility), effectively permitting configuration descriptions to be automatically checked for structural consistency.
Reflective middleware framework integrated with the meta-model

This thesis has proposed a novel approach for reflective middleware, based on the definition of the meta-level architecture on top of meta-information management. This enables the benefits of configurability and adaptability to be equally applied to the reflection mechanisms. In particular, the definition of meta-objects in terms of the meta-model permits the seamless extension of the reflection capabilities, also enabling the co-existence of alternative meta-object protocols. In addition, the use of the meta-model constructs as the basis for reification enhances the expressive power of the MOPs, as all elements employed in the configuration of a platform can be exposed through reflection. This also contributes to the uniformity of the concepts used for configuration and reconfiguration, greatly simplifying the whole architecture.

In addition, as a consequence of the integrated approach, the thesis has proposed the use of the concept of type evolution to enable new configuration descriptions to be generated from the results of dynamic adaptation. As demonstrated, the technique has a positive impact on the development process, as it permits the efforts spent in adaptations to be seamlessly reused in different circumstances.

Finally, although the combination of reflection and meta-information management has been proposed in the context of reflective middleware, the author argues that the overall approach is sufficiently generic to be applied in other contexts with similar features and requirements. This therefore represents a valuable contribution to the more general field of reflection and meta-level architectures, although further research would be required to properly generalise the several elements of the approach.

8.3.2 Other significant results

Design principles for reflective middleware

The study of the fundamentals of reflection and their application to middleware has enabled the identification of a set of design principles for the different aspects of reflective middleware (as introduced in Chapter 2, section 2.4.2). This is a timely contribution, as currently there is no wide agreement on the exact meaning of reflection for middleware and what features a reflective middleware platform should provide. The design principles therefore aim to contribute to a better understanding of the field and to its more precise definition, besides also serving as a framework for
evaluating current and future reflective middleware platforms (such as done in section 8.4 regarding the Meta-ORB architecture).

**Extension to the CORBA meta-model**

The Meta-ORB meta-model has been proposed as an extension to the CORBA meta-model, inspired by RM-ODP, in order to support an open platform infrastructure. Central concepts of the meta-model include first-class components and explicit binding objects, as well as support for hierarchical composition, media type and QoS specification and stream interaction. Importantly, the same meta-model applies to application configuration as well, contributing to the uniformity of the several aspects of a system. It is hoped that this work will prove helpful to understand how CORBA can be enhanced to offer more flexible support for applications.

**Extensible and distributed framework for explicit binding**

This thesis has also contributed to the design of binding frameworks, which are an important aspect of multimedia middleware platforms. An optimised distributed binding protocol has been proposed, which complements the Meta-ORB meta-model and provides a concrete realisation of the abstract binding establishment process recommended in RM-ODP [ITU-T/ISO 1998b]. The protocol defines binding establishment as a cooperative process, involving independent local binding factories to instantiate each of the endpoints of a binding, thus improving the overall efficiency. In addition, the design of binding factories has been proposed in a generic way, thus eliminating the need to define a particular factory for each new binding type.

**Meta-object protocols for structural reflection**

As a means to demonstrate the approach proposed in the thesis, meta-object protocols have been defined to populate the structural meta-space models. Such MOPs include a representative set of operations for the inspection of interfaces, components and binding objects, as well as for the adaptation of composite components and bindings. The implementation of the MOPs has also been provided, in terms of meta-object components that realise the reification and causal-connection mechanisms. Overall, this represents a contribution to the understanding of how reflection can be concretely used to support implementation openness in middleware.
8.4 Requirements revisited

A major aim of the design principles introduced in Chapter 2 is to serve as an evaluation framework for reflective middleware platforms, showing the degree to which important issues are dealt with. Table 8.1 presents the results of applying this assessment to the Meta-ORB architecture, followed by some overall conclusions.

Table 8.1 – Comparing the Meta-ORB reflective middleware architecture with the requirements and design principles introduced in Chapter 2

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Meta-ORB Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular platform infrastructure</td>
<td>This principle has been achieved through the use of a component-based meta-model, which allows the different elements and aspects of a platform configuration to be individually identified and manipulated.</td>
</tr>
<tr>
<td>Language and system independence</td>
<td>This requirement has been met by adoption of a language-neutral meta-model as the basis to define the MOP.</td>
</tr>
<tr>
<td>Approach to separation of concerns</td>
<td>A clear cut is made between base- and meta-level, where the inherent functionality of both middleware and applications is considered at the base-level, while the facilities for their handling are at the meta-level.</td>
</tr>
<tr>
<td>Access to the meta-level</td>
<td>Only explicit access has been provided, due to the fact that only structural reflection has been considered in the design.</td>
</tr>
<tr>
<td>Granularity</td>
<td>Reflection can be applied to interfaces, components and binding objects (although adaptation only applies to the latter two). However, due to the support for composition in the meta-model, the actual granularity can be chosen from simple primitive entities to entire platform configurations.</td>
</tr>
<tr>
<td>Scope of reflection</td>
<td>A hybrid approach has been adopted, due to the combination of object-oriented reflection and meta-information management. While the former allows reflective access based on single object instances (for both inspection and adaptation), the latter enables inspection access on a type basis, thus applying to all instances of a type.</td>
</tr>
<tr>
<td>Pervasiveness of reflection</td>
<td>The common use of the meta-model by the configuration and reflection mechanisms enables all aspects used in the construction of a platform to be subject to reflection.</td>
</tr>
<tr>
<td>Uniformity of the reflection model</td>
<td>This has been achieved, as the same overall MOP applies to all levels of granularity and to all elements of a platform's design. However, it has not been achieved with respect to the different scopes of reflection, since type-based introspection is not performed through the MOP.</td>
</tr>
<tr>
<td>Configurability and re-configurability</td>
<td>A unified approach has again been achieved through the use of a common meta-model, since the meta-types it defines are used as the basis to define the facilities for both configuration and reconfiguration.</td>
</tr>
<tr>
<td>Safety of reflection</td>
<td>This aspect was not a major aim in this thesis and is considered an issue for future work (see 8.5.5).</td>
</tr>
<tr>
<td>Management of meta-level complexity</td>
<td>This has been achieved by the adoption of a multi-model reflection framework (although only applied to structural reflection).</td>
</tr>
<tr>
<td>Explicit management of meta-information</td>
<td>This was a central requirement in this thesis and motivated the novel approach to integrate reflection and meta-information management.</td>
</tr>
</tbody>
</table>
| Performance-enhancing mechanisms                  | This issue has not been particularly considered in this thesis. However, Chapter 7 has presented some preliminary ideas for simple meta-level optimisations. In addition, the configurability of the architecture enables particular platform instances to be optimised for performance.
As can be seen from the table, most of the design principles and requirements have been thoroughly considered, although important gaps still exist regarding the issues of safety and performance, as well as the different modes of meta-level access. In light of this, it can be concluded that this thesis has achieved the goal of presenting a comprehensive reflective middleware architecture, especially when compared with existing related work in the area (see analysis in Chapter 2).

8.5 Future work

8.5.1 Meta-model extensibility

The explicit treatment given to meta-types in the MOF enables new meta-types to be introduced into an existing meta-model. In a middleware meta-model, this could be used to extend the set of constructs that the platform is able to handle. As an example, the Meta-ORB meta-model could be augmented with meta-types defining different notations to specify QoS-related meta-information. This is a promising approach for representing constructs of this kind, as it is often difficult to devise a single notation that is appropriate in all cases. Thus, an important area for future work is to investigate the use of the MOF and associated technologies (such as XMI and OCL) to define and represent meta-types so that they can be meaningfully introduced into an existing meta-model, especially considering their semantics. In addition, the feasibility of introducing new meta-types at runtime, without having to reinitialise the meta-model needs to be investigated, given the MOF-based tools currently available. Note that such future work will require the use of the MOF-based implementation of the Type Repository (described in Chapter 6).

8.5.2 Interoperability between platforms with different meta-models

In a similar way, treating meta-types as first-class entities enables a promising approach to interoperability, which demands a closer investigation. In particular, the dynamic interpretation of meta-types makes it possible for a platform to properly deal with constructs, such as interactions and argument types, defined according to a foreign meta-model. In practice, a bridge can be defined at the meta-model level, in order to perform the translation of foreign interactions in a transparent way. While it has been argued that this kind of approach may simply move the problem to the next
level of abstraction [Thompson 1998], due to the heterogeneity of (meta-)meta-
models, the current widespread acceptance of the MOF as a universal meta-modelling
architecture makes such argument less relevant. Nevertheless, it must be noticed that a
completely automated process may not be feasible, especially due to semantic
differences between meta-models and to the need to properly map primitive types.

8.5.3 Extend the approach to behavioural reflection

The approach presented in this thesis concentrated on the structural aspects of the
meta-level. However, the overall principles of this approach apply to behavioural
reflection as well. In particular, the meta-information handled by the Resources meta-
space model can be managed through a meta-information repository. In this respect,
related work at Lancaster has focused on the specification of meta-information
defining resources and their allocation, using the concept of interaction tasks [Duran-
Limon and Blair 2000b]. To complement this, future work could investigate the
definition of explicit meta-types for resource-related meta-information. Among other
things, this would enable the dynamic derivation of new task definitions, as a result of
the reflective adaptation of existing tasks (in an approach similar to type evolution).

8.5.4 Investigate the use of type evolution constraints

Another important aspect of the integration between reflection and meta-
information management refers to the use of rules to constrain reflective
reconfiguration and help guaranteeing the consistency and usefulness of adapted
configurations. In this respect, though, further research is required to demonstrate the
effectiveness of the approach. This would involve the definition of useful rules, such
as architectural constraints, as well as their effective representation and the necessary
mechanisms for their enforcement. Note that the support for flexible type evolution
constraints would require an extension to the current meta-model, in order to enable
the specification of constraints and their association with particular type definitions.

8.5.5 Safety and security of reflection

The issue of safety refers to the need to preserve the consistency of the platform, in
the presence of adaptation. A promising answer to this is the proposed use of type
evolution constraints, although, as seen in Chapter 6, this approach does not
encompass all kinds of adaptation. A complementary solution has to do with reconfiguration failures and refers to the implementation of adaptation operations in a transactional way, so that the whole configuration can revert to the original state in case of a failure. On the other hand, regarding security, it is important to prevent the malicious or unauthorised use of reconfiguration. A general solution to this would be to improve the programming model with support for access control, so that critical interfaces, including those used to provide the MOPs, can have a level of protection.

8.5.6 Improvements to the binding framework

As noted in Chapter 7, enhancements are required to further simplify the task of binding definition. Firstly, facilities could be provided to automatically generate the type and implementation of stub components, based on the particular target interface types. Secondly, separate meta-types could be introduced to deal with the type and template aspects of binding definition, in order to facilitate the reuse of binding configurations (currently, these two aspects are merged in the CompBindingDef meta-type). In this way, although binding types would still have to be defined on a per application basis (as they would deal with the external aspects of a binding, such as the target interface types), binding templates would typically be reusable. Notably, a given binding type would refer to a particular template to provide the appropriate configuration. It is hoped that future work in this area will contribute to an approach for the reusability of binding definitions in an off-the-shelf manner, similarly to what has been achieved in the area of component-based software development.

8.5.7 Enhanced tool support

The usability of the programming environment, as noted in Chapter 7, could be substantially improved by the availability of appropriate interactive tools for the tasks of configuration (especially for component and binding definition) and adaptation. In particular, the use of graphical user interfaces for this purpose can be of great help, in order to enable such tasks to be performed in a more intuitive manner (note that the GUI tool currently available, although supporting configuration, is based on textual definitions). It is also important that future work in this area considers the explicit treatment given to meta-information in the approach (since such tools would basically provide an interactive way to handle meta-information). In this respect, the generic design of interactive tools should be investigated, based on the dynamic interpretation
of MOF-defined meta-types, instead of hard-coding the meta-type definitions in the implementation of the tools. Note that such work would further highlight the benefits of using an explicit meta-information architecture, which enables the design and implementation of meta-model independent tools and facilities.

8.6 Concluding remarks

This thesis has concentrated on middleware support for distributed applications with dynamic requirements, such as those that involve multimedia and mobility. The work has observed that current middleware technologies, although offering an appropriate level of abstraction and transparency, do not provide the kind of customisable and adaptable support that such applications need. The author envisages that the increasing importance of this category of applications will drive the development of more flexible middleware architectures. This has already been observed by the trend to introduce customisation options in current middleware standards. However, the author also believes that a more principled, as opposed to ad hoc, approach should be adopted, in order to extend the benefits of openness to all aspects of a platform, in a uniform and effective way. In this thesis, the author has proposed a middleware architecture that combines the adaptability enabled by reflection techniques with the configurability facilitated by the use of meta-information management. It is hoped that the proposed approach will help to influence the development of more flexible middleware technologies and standards that are capable of matching the requirements of current and future applications.
Appendix A  Related Work on Reflective Middleware: Comparative Analysis

A.1 Introduction

The following tables present an analysis of the features of the reflective middleware platforms examined in Chapter 2, according to the evaluation framework discussed in that chapter.

<table>
<thead>
<tr>
<th></th>
<th>1) Modular platform structure</th>
<th>2) Language and system independence</th>
<th>3) Approach to separation of concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlexiNet (original binding framework)</td>
<td>the binding framework is structured in terms of a protocol stack</td>
<td>not provided; depends on extended reflection facilities of Java</td>
<td>the upper protocol layer is the equivalent of a meta-space</td>
</tr>
<tr>
<td>OpenCORBA</td>
<td>not explicitly defined</td>
<td>not provided; depends on the reflection capabilities of NeoClassTalk</td>
<td>aspects of the platform functionality are regarded as part of the meta-level</td>
</tr>
<tr>
<td>2K, dynamicTAO and UICTM</td>
<td>the ORB is structured in terms of a component model</td>
<td>based on a language-independent component model and associated meta-interfaces</td>
<td>the platform components can be seen as base-level entities, relatively to their respective meta-objects</td>
</tr>
<tr>
<td>DART</td>
<td>implicit in the meta-level structure (in terms of meta-objects, reflectors, method selectors and the DART manager)</td>
<td>not provided; depends on the reflection facilities of Open C++ v2</td>
<td>meta-objects (for reflective methods) are part of the data path; reflectors play the role of meta-meta-objects</td>
</tr>
<tr>
<td>RORB</td>
<td>a fixed set of aspects is identified, and each one is implemented separately from the others</td>
<td>based on language-independent meta-interfaces</td>
<td>meta-objects implement the underlying aspects of the platform (and provide the meta-interfaces for their control)</td>
</tr>
<tr>
<td>Quarterware</td>
<td>a fixed set of aspects is identified, in terms of the meta-interfaces to manipulate them</td>
<td>based on language-independent meta-interfaces</td>
<td>meta-interfaces are provided by first-class objects that are part of the ORB implementation</td>
</tr>
<tr>
<td>mChaRM</td>
<td>the communication aspect is implemented separately from other aspects</td>
<td>based on a language-independent API provided by the channels</td>
<td>communication channels are considered as the meta-level</td>
</tr>
</tbody>
</table>
### Table A.3 – Reflective middleware comparative analysis (cont.)

<table>
<thead>
<tr>
<th></th>
<th>4) Access to meta-level</th>
<th>5) Granularity flexibility</th>
<th>6) Scope of reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FlexiNet</strong>&lt;br&gt;(original binding framework)</td>
<td><em>implicit</em>: request interception; <em>explicit</em>: meta-object (interceptor) replacement</td>
<td>fixed: each meta-object corresponds to an aspect of a protocol layer</td>
<td>a binding between two objects</td>
</tr>
<tr>
<td><strong>OpenCORBA</strong></td>
<td><em>explicit</em>: change the meta-class of a class, using the NeoClassTalk MOP</td>
<td>fixed: at the level of specific ORB features (although meta-classes can introduce changes at a finer granularity)</td>
<td>all instances of the same proxy or type checker class</td>
</tr>
<tr>
<td><strong>2K, dynamicTAO and UIC™</strong></td>
<td><em>explicit</em>: complex MOP to handle the structure of the platform; <em>hybrid</em>: monitors automatically call the meta-objects</td>
<td>flexible: arbitrarily complex components and their dependencies can be manipulated (although only coarse-grained components, in practice)</td>
<td>individual component instances</td>
</tr>
<tr>
<td><strong>DART</strong></td>
<td><em>implicit</em>: reflectors and method selectors redirect method calls; <em>hybrid</em>: events cause adaptation; <em>explicit</em>: runtime installation of adaptation policies</td>
<td>fixed: individual non-functional aspects of a method (reflective methods); individual method implementations (adaptive methods)</td>
<td>per object or per method</td>
</tr>
<tr>
<td><strong>RORB</strong></td>
<td><em>explicit</em>: access to the interface of meta-objects for their adaptation</td>
<td>fixed: at the level of specific mechanisms of the ORB and applications</td>
<td>per object (application-level meta-objects) and per ORB instance (system meta-objects)</td>
</tr>
<tr>
<td><strong>Quarterware</strong></td>
<td><em>explicit</em>: via the ORB’s meta-interfaces; <em>hybrid</em>: auto-adaptation through user-provided policies</td>
<td>fixed: at the level of individual mechanisms of the ORB</td>
<td>per ORB instance</td>
</tr>
<tr>
<td><strong>mChaRM</strong></td>
<td><em>explicit</em>: channel provides interface for introspection and intercession; <em>implicit</em>: message interception</td>
<td>fixed: at the level of communication channels</td>
<td>per communications session</td>
</tr>
</tbody>
</table>
### Table A.4 – Reflective middleware comparative analysis (cont.)

<table>
<thead>
<tr>
<th>Framework</th>
<th>7) Pervasiveness of reflection</th>
<th>8) Uniformity of the reflection model</th>
<th>9) Configurability and re-configurability</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlexiNet (original binding framework)</td>
<td>reflection is limited to the aspects of the protocol layers that compose a binding between two objects</td>
<td>non-uniform: in the upper layer, APM’s Reflective Java MOP; in other layers all layers: basic Java Core Reflection (only Java introspection)</td>
<td>static configuration based on default options and policies; dynamic reconfiguration by replacing meta-objects at the upper layer (only)</td>
</tr>
<tr>
<td>OpenCORBA</td>
<td>a limited range of aspects can be reified (proxies, type checkers and the IR exception mechanism)</td>
<td>uniform: NeoClassTalk MOP applies to all class–meta-class links (for all aspects)</td>
<td>the link between a class and its meta-class can be set at compile-time and may be altered at runtime</td>
</tr>
<tr>
<td>2K, dynamicTAO and UICTM</td>
<td>the same reflective facilities can be applied to any component of the ORB (at the adopted level of granularity)</td>
<td>semi-uniform: component configurators define a MOP framework, which can be further specialised for particular kinds of base objects</td>
<td>explicit facilities for static configuration, plus the dynamic management of component dependencies; both based on the same framework</td>
</tr>
<tr>
<td>DART</td>
<td>apart from method selection and interception, other ORB aspects cannot be reified</td>
<td>non-uniform: OpenC++ MOP at compile-time; the DART manager enables runtime adaptation</td>
<td>reflective and adaptive methods can be statically configured using OpenC++; they can be dynamically reconfigured using adaptation policies of the DART manager</td>
</tr>
<tr>
<td>RORB</td>
<td>limited to a fixed set of ORB and application mechanisms</td>
<td>non-uniform: each aspectual meta-object introduces its own meta-interface</td>
<td>meta-objects can be installed either statically or dynamically</td>
</tr>
<tr>
<td>Quarterware</td>
<td>limited to a fixed set of ORB mechanisms</td>
<td>non-uniform: each reified aspect has its own specific meta-interface</td>
<td>static: framework specialisation; meta-interfaces for dynamic set up of ORB mechanisms</td>
</tr>
<tr>
<td>mChaRM</td>
<td>only aspects of the communications channel can be reified</td>
<td>uniform: a two-part MOP is defined to deal with all kinds of messages and channels</td>
<td>channel installed statically; dynamic change of channel behaviour</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>-----------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>FlexiNet (original binding framework)</td>
<td>based on type checking of meta-objects</td>
<td>two separate aspects: binding behaviour and message structure</td>
<td>not defined</td>
</tr>
<tr>
<td>OpenCORBA</td>
<td>not defined</td>
<td>distinct ORB aspects are reified separately</td>
<td>not defined</td>
</tr>
<tr>
<td>2K, dynamicTAO and UICTM</td>
<td>ORB consistency management; access control to MOP; sandboxing of dynamic code</td>
<td>different classes of meta-objects to reify distinct kinds of components of the platform</td>
<td>not defined</td>
</tr>
<tr>
<td>DART</td>
<td>potential side effects are localised to particular methods</td>
<td>distinct meta-level features to deal with structure and behaviour</td>
<td>not defined</td>
</tr>
<tr>
<td>RORB</td>
<td>side effects can be limited to single objects (app.-level meta-objects)</td>
<td>distinct ORB aspects are reified separately</td>
<td>not defined</td>
</tr>
<tr>
<td>Quarterware</td>
<td>not defined</td>
<td>distinct ORB aspects handled via separate meta-interfaces</td>
<td>not defined</td>
</tr>
<tr>
<td>mChaRM</td>
<td>side effects are limited to a single communications channel</td>
<td>not defined</td>
<td>not defined</td>
</tr>
</tbody>
</table>
Appendix B  Specifications of the Meta-ORB meta-model

B.1  Introduction

This appendix accompanies Chapter 4, and provides the definitions related to the Meta-ORB meta-model, notably its MOF-based specification in the MODL language, and the grammar of the of the Meta-ORB ODL language, which provides a concrete syntax for expressing model elements. In addition, the appendix also presents examples of the use of the ODL language.

B.2  MODL description of the Meta-ORB meta-model

This section presents the complete specification of the Meta-ORB meta-model, in terms of the MODL (Meta-Object Definition Language), which is used by the dMOF tool suite [DSTC 2001]. The specification corresponds to the UML diagrams presented in Chapter 4, although with a few changes made to avoid the circularity of package definitions. In particular, a top-level package, called MetaORB, is defined so that the other packages are nested inside it. This packages also takes up the role of the original BaseIDL package. In addition, note that the BasicTypes package is not required, since MODL directly uses the primitive types (and also typecodes) defined in CORBA 2.0. Importantly, the specification presented here also includes the specific operations to create meta-information elements, as well as the constraints associated with the meta-model elements. Regarding the latter, although OCL [OMG 2000d] is the de facto standard for formalising MOF constraints, the meta-model constraints currently defined here are presented informally, in natural language\(^1\). The formal definition of these constraints in OCL is a subject for future work. Furthermore, note that the constraints currently defined are mainly related to containment rules. The identification of constraints involving other semantic aspects is also an issue for future work.

\(^1\) In addition, note that although MODL enables the specification of constraints as part of a meta-model, the current release of the dMOF tool (in particular, the MOFlet generator) does not generate code for them.
/** Root package **
package MetaORB{

// Exception raised by operations in the type repository.
exception TRException {
  short error_code;
  string reason;
};

// These values qualify the kind of all modelling constructs in the repository.
enum DefinitionKind{dk_none, dk_all, dk_Attribute, dk_Constant, dk_Exception,
dk_Interface, dk_Module, dk_Operation, dk_Typedef,
dk_Alias, dk_Struct, dk_Union, dk_Enum, dk_Primitive,
dk_String, dk_Sequence, dk_Array, dk_Repository,
dk_Wstring, dk_Fixed,
dk_Flow, dk_StrInterface, dk_Signal, dk_SigInterface,
dk_GenericMediaType, dk_MediaTypeSystem,
dk_SpecificMediaType, dk_MediaSpecification,
dk_PrimComponent, dk_Component, dk_PrimRole,
dk_Role, dk_PrimBinding, dk_Binding};

// Abstract class which is the base for all (meta-)model elements.
// (This class is not strictly needed in a MOF world, since all its features can
// be derived from the MOF server or its associated interfaces).
abstract class IRObject {
  // The value for this attribute is derived from the MOF Server's
  // information describing the type of the meta-object in question.
  readonly derived attribute DefinitionKind def_kind;

  // Operation to destroy a model element. It is implemented in terms of
  // MOF operations on the related class (e.g. refDelete, inherited from
  // Reflective::RefObject) and the remove operations generated for any eventual
  // associations.
  void destroy();
};

// Forward declarations.
singleton class Repository;
abstract class Container;
class PrimitiveDef;
class ModuleDef;
class StructDef;
class UnionDef;
class ModuleDef;

abstract class Contained : IRObject {
  // The name of the model element.
  attribute string name;

  // The repository id of the model element.
  attribute string id;

  // The version of the model element.
  attribute string version;

  // The absolute (scoped) name of the model element.
  readonly derived attribute string absolute_name;

  // The repository that contains this model element. (This points to the artefact
  // class Repository - actually, the containing repository is the MOF server).
  readonly derived attribute Repository containing_repository;

  // Reference to the container of this model element.
  reference defined_in to containerElement of Contains;

  // Operation to move this model element to another container.
  void move(in Container new_container, in string new_name, in string new_version);
};

// Artefact abstract class introduced in the MOF model to be the base class for all
// model elements that have an intrinsic type.
abstract class Typed {
  // Attribute giving the typecode of a typed model element (it is derived from
  // the referenced IDLType, from the attribute with same name in that class).
  readonly derived attribute TypeCode type_code;
Appendix B – Specifications of the Meta-ORB meta-model

// Reference to the IDLType that is the type of this typed model element.
reference type_def to idlType of TypedBy;

// Abstract class representing which is the base for all model elements that
// constitute
// types (i.e. that can be referred to as the type for another model element).
abstract class IDLType : IRObject {
    // The type code of an IDLType. It is derived by the sub-classes of this class by
    // creating a specific TypeCode object.
    readonly derived attribute TypeCode type;

    // Class representing constant model elements.
    class ConstantDef : Typed, Contained {
        readonly attribute any value;

        constraint TypeMustBePrimitive "Eng":
            "[C-01] The type of a constraint must be a primitive type"
    }

    // Abstract class representing non-anonymous model elements that constitute the types
    // of other model elements.
    abstract class TypedefDef : Contained, IDLType {

        // Class representing a member (field) of a struct.
        class StructMember : Typed {
            attribute string name;
        }
    }

    typedef sequence<StructMember> StructMemberSeq;

    // Class representing a member of a union.
    class UnionMember : Typed {
        attribute string name;
        attribute any label;
    }
    typedef sequence<UnionMember> UnionMemberSeq;

    // Class representing enumeration definitions.
    class EnumDef : TypedefDef {
        attribute ordered set [1..*] of string members;
    }
    typedef sequence<string> EnumMemberSeq;

    // Class representing aliases definitions.
    class AliasDef : Typed, TypedefDef {
        // The original type of the alias (this is directly derived from the type_def
        // reference inherited from Typed). It is provided here to preserve compatibility
        // with the standard attribute name defined in the CORBA IR.
        derived attribute IDLType original_type_def;
    }

    // Enumeration of all possible kinds of primitive types (used to qualify a
    // PrimitiveDef model element).
    enum PrimitiveKind {pk_null, pk_void, pk_short, pk_ushort, pk_long, pk_ulong,
        pk_float, pk_double, pk_boolean, pk_char, pk_octet,
        pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
        pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
        pk_iref};

    // Class representing primitive types.
    class PrimitiveDef : IDLType {
        readonly attribute PrimitiveKind kind;
    }

    // Class representing bounded strings.
    class StringDef : IDLType {
        attribute unsigned long bound;
    }

    // Class representing wide strings.
    class WstringDef : IDLType {
        attribute unsigned long bound;
    }
Appendix B – Specifications of the Meta-ORB meta-model

// Class representing Fixed point types.
class FixedDef : IDLType {
    attribute unsigned short digits;
    attribute short scale;
};

// Class representing sequence types.
class SequenceDef : Typed, IDLType {
    attribute unsigned long bound;
    // Derived attributes representing the typecode and the IDLType that give the
    // type of the sequence’s elements. (these are provided for name conformance
    // with the standard attribute names in the CORBA IR):
    readonly derived attribute TypeCode element_type;
    // derived from attribute type, inherited from Typed.
    derived attribute IDLType element_type_def;
};

// Class representing array types.
class ArrayDef : Typed, IDLType {
    attribute unsigned long length;
    // Derived attributes representing the typecode and the IDLType that give the
    // type of the array’s elements. (these are provided for name conformance with
    // the standard attribute names in the CORBA IR):
    readonly derived attribute TypeCode element_type;
    // derived from attribute type, inherited from Typed.
    derived attribute IDLType element_type_def;
};

// Abstract class representing the functionality of model elements
// that may contain other model elements.
abstract class TopContainer {
    // Operation to lookup for a contained object given its scoped name.
    set [0..*] of Contained lookup_name(in string search_name,
        in long levels_to_search,
        in DefinitionKind limit_type,
        in boolean exclude_inherited);

    // Operation to lookup for a contained object given its non-scoped name.
    Contained lookup(in string search_name);

    // Operation to return all the contents of this Container. The list of results
    // may be filtered by the two arguments, limit_type and exclude_inherited. This
    // operation is derived from the allContents reference, which projects the
    // container_element end of association Contains.
    set [0..*] of Contained contents(in DefinitionKind limit_type,
        in boolean exclude_inherited);

    // Auxiliary reference to give attribute-like access to the Contains association
    // from the Container side. (This reference would better be defined with private
    // visibility, since, for reasons of compatibility with the standard IR
    // interfaces, it may only be accessed by the implementation of operation
    // contents.)
    reference allContents to containedElement of Contains;

    // Create-like operations (used by several kinds of Containers).
    ModuleDef create_module(in string id, in string name, in string version);
    ConstantDef create_constant(in string id, in string name, in string version,
        in IDLType type, in any value);
    StructDef create_struct(in string id, in string name, in string version,
        in IDLType type, in any value);
    UnionDef create_union(in string id, in string name, in string version,
        in IDLType discriminator_type, in UnionMemberSeq members);
    EnumDef create_enum(in string id, in string name, in string version,
        in IDLType type, in any value, in EnumMemberSeq members);
}
Appendix B – Specifications of the Meta-ORB meta-model

AliasDef create_alias(in string id, in string name, in string version, in IDLType original_type);

// Note: Other kinds of Contained objects can only be created inside modules or inside the more specific Containers.

// Base class for all containers (except for the root container).
abstract class Container : TopContainer, Contained {

// Class representing struct definitions.
class StructDef : TypedefDef, Container {
    attribute ordered set [1..*] of StructMember members;

// Class representing union definitions.
class UnionDef : TypedefDef, Container {
    attribute ordered set [1..*] of UnionMember members;

    // The union's discriminator type (it should be derived from the discriminator_type_def reference, which points to the IDLType that is the type of the discriminator).
    readonly attribute TypeCode discriminator_type;

    // Reference to the IDLType that is the type of the union's discriminator (through the DiscriminatedBy reference).
    reference discriminator_type_def to discriminatorTypeDef of DiscriminatedBy;

// ** Associations **
association Contains {
    composite end single TopContainer containerElement;
    end set [0..*] of Contained containedElement;
}

association TypedBy {
    end set [0..*] of Typed typed;
    end single IDLType idlType;
}

association DiscriminatedBy {
    end set [0..*] of UnionDef unionDef;
    end single IDLType discriminatorTypeDef;
}

// ** Nested package **
package MediaDefs{

    // Enumeration of the valid major media types.
    enum MajorMediaTypes {
        MEDIA_AUDIO, MEDIA_VIDEO, MEDIA_ANIMATION, MEDIA_IMAGE
    }

    // Enumeration of the valid kinds for media attributes.
    enum MediaAttrValueKind {
        VALUE_SINGLE, VALUE_RANGE, VALUE_SEQUENCE
    }

    // Class representing generic media attributes.
class GenericMediaAttrDescription : ::MetaORB::Typed {
        attribute string name;
        attribute MediaAttrValueKind value_kind;
        attribute ordered set [1..*] of any values;
        derived attribute TypeCode att_base_type;

        constraint AttrBaseTypeMustBeNumeric "Eng" :
            "[C-02] The type of the attribute values must be a numeric type"
    }

typedef sequence <GenericMediaAttrDescription> GenericMediaAttrList;

    // Forward declaration.
class SpecificMediaTypeDef;

// Enumeration of the valid qualifier modes for the values of specific media attributes.
enum QualifierMode {QUALIF_NONE, QUALIF_MAXIMISE, QUALIF_MINIMISE};

// Structure representing specific media attributes.
struct SpecificMediaAttrDescription{
    string name;
    MediaAttrValueKind value_kind;
    QualifierMode qualifier;
    sequence <any> values;
};
typedef sequence<SpecificMediaAttrDescription> SpecificMediaAttrList;

// Class representing generic media type definitions.
class GenericMediaTypeDef : ::MetaORB::Contained {
    readonly attribute MajorMediaTypes major_media_type;
    attribute set [1..*] of GenericMediaAttrDescription attrs;
    // Operation to create a specific media type from this generic media type.
    SpecificMediaTypeDef specialise(in string id, in string name, in string version, in ::MetaORB::Container defined_in, in MajorMediaTypes major_media_type, in SpecificMediaAttrList spec_attr_list) raises (::MetaORB::TRException);
    constraint GenMediaTypeContainment "Eng" :
        "[C-03] A GenericMediaTypeDef an only be defined in a MediaTypeSystemDef";
};

// Class representing media type system definitions.
class MediaTypeSystemDef : ::MetaORB::Container {
    GenericMediaTypeDef create_generic_media_type(in string id, in string name, in string version, in MajorMediaTypes major_media_type, in GenericMediaAttrList attrs);
    constraint MediaTypeSystemContainment "Eng" :
        "[C-04] A MediaTypeSystemDef can only be defined in the root Repository and there can only be a single instance of it.";
};

// Class representing specific media type definitions.
class SpecificMediaTypeDef : ::MetaORB::Contained {
    readonly attribute MajorMediaTypes major_media_type;
    attribute set [1..*] of SpecificMediaAttrDescription attrs;
    // Reference to the generic media type definition that defines the encoding format for this specific media type.
    reference encoding to genericMediaType of GenericMediaType;
    constraint SpecMediaTypeContainment "Eng" :
        "[C-05] A SpecificMediaTypeDef can only be defined in a MediaSpecificationDef";
};

// Class representing media specification definitions.
class MediaSpecificationDef : ::MetaORB::TypedefDef, ::MetaORB::Container {
    readonly attribute MajorMediaTypes major_media_type;
    SpecificMediaTypeDef create_specific_media_type(in string id, in string name, in string version, in MajorMediaTypes major_media_type, in string encoding, in SpecificMediaAttrList attrs);
    constraint MediaSpecContainment "Eng" :
        "[C-06] A MediaSpecificationDef can only be defined in an InterfaceDef";
};

// ** Associations **
association GenericMediaType {
    end set [0..*] of SpecificMediaTypeDef specificMediaType;
Appendix B – Specifications of the Meta-ORB meta-model

end single GenericMediaTypeDef genericMediaType;
);
}; // ** End of nested package MediaDefs **

// ** Nested package **
package QoSDefs {

// Enumeration representing the valid kinds of QoS attributes.
enum QoSAttrValueKind {QOS_TARGET, QOS_RANGE_MAX, QOS_RANGE_MIN, QOS_MAX, QOS_MIN};

// Enumeration representing the valid units of measurement for QoS attributes.
enum MeasureUnits {UNIT_ABS, UNIT_PERCENT, UNIT_MSEC, UNIT_SEC, UNIT_B, UNIT_KB, UNIT_MB, UNIT_BPS, UNIT_KBPS, UNIT_MBPS};

// Class representing QoS attribute descriptions (the declarations
// of valid QoS attributes that are stored in the QoS attribute bases).
class QoSCharacteristic : { attribute string name; attribute QoSAttrValueKind value_kind; attribute MeasureUnits unit; }; typedef sequence<QoSCharacteristic> QoSCharacteristicSeq;

// Class representing QoS attribute bases.
class QoSBaseDef : ::MetaORB::Contained { attribute set [1..*] of QoSCharacteristic attr_descrs; constraint QoSBaseContainment "Eng" :
"[C-07] A QoSBaseDef can only be defined in the root Repository and there can be only a single instance of it at any time.";
};

// Structure representing a specific occurrence of a QoS attribute
// (i.e. in a QoS annotation).
struct QoSAttribute{ string name; double value_min; double value_max; double value_target; }; typedef sequence<QoSAttribute> QoSAttributeSeq;

// Class representing QoS annotations.
class QoSAnnotationDef : ::MetaORB::IRObject { attribute set [1..*] of QoSAttribute attrs; }
}; // ** End of nested package QoSDefs **

// ** Nested package **
package Interfaces {

// Class representing exception definitions.
class ExceptionDef : ::MetaORB::Container { // The type of the exception (a tk_except is created and returned // when accessing this attribute - it does not need to be stored) readonly derived attribute TypeCode type; attribute set [0..*] of ::MetaORB::StructMember members; }
typedef sequence<ExceptionDef> ExceptionDefSeq;

// Enumeration of the valid attribute modes.
enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

// Class representing attribute definitions.
class AttributeDef : ::MetaORB::Typed, ::MetaORB::Contained { attribute AttributeMode mode;
constraint AttrContainment "Eng" :
"[C-08] An AttributeDef can only be defined in an InterfaceDef.";
};

// Class representing the base interface for interface definition model elements.
abstract class InterfaceDef : ::MetaORB::Container, ::MetaORB::IDLType { 223}
// Operation to compare this interface with another one.
boolean is_a(in string interface_id);

// Reference to the base interfaces of this interface.
reference base_interfaces to base of BaseInterfaces;

AttributeDef create_attribute(in string id, in string name,
in string version,
in ::MetaORB::IDLType type,
in AttributeMode mode);

::MetaORB::MediaDefs::MediaSpecificationDef create_media_specification(
in string id, in string name, in string version,
in string major_media_type);

constraint SupertypeKindMustBeSame "Eng" :
"[C-09] Base Interfaces must be of the same interface style";
constraint InterfContainment "Eng" :
"[C-10] An InterfaceDef must be defined within the a ModuleDef";

// Artefact (new) class which generalises all meta-model elements that
// may have related QoS annotations.
abstract class QoSConstrained {
// This attribute *should* be derived from the reference
readonly attribute boolean qos_constrained;
// Reference to the constraining QoS annotation.
reference qos_annotation to qosAnnotation of QoS;
}

// Enumeration of the valid parameter modes.
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};

// Class representing operation parameters.
class ParameterDescription : ::MetaORB::Typed {
  attribute string name;
  attribute ParameterMode mode;

  constraint ParamTypeMustNotBeContinuous "Eng" :
"[C-23] The type of a parameter must not be a continuous media type.";
};
typedef sequence<ParameterDescription> ParameterDescriptionSeq;

// Enumeration of the valid operation modes.
enum OperationMode {OP_NORMAL, OP_ONEWAY};

// Enumeration of valid operation causalities.
enum OperationCausality {OP_PROVIDED, OP_REQUIRED};
typedef sequence<string> stringSeq;

// Class representing operation definitions.
class OperationDef : ::MetaORB::Typed, ::MetaORB::Contained, QoSConstrained {
  // Derived attributes representing the typecode and the IDLType that give
  // the type of the operation's result. (these are provided for name
  // conformance with the standard attribute names in the CORBA IR):
  readonly derived attribute TypeCode result;

  derived attribute ::MetaORB::IDLType result_def;

  attribute ordered set [0..*] of ParameterDescription params;
  attribute OperationMode mode;
  attribute ordered set [0..*] of stringSeq contexts;
  attribute OperationCausality causality;

  // Reference giving access to the exceptions raised by this operation
  // reference exceptions to exceptionDef of CanRaise;

  constraint OpDefContainment "Eng" :
"[C-11] An OperationDef can only be defined in an OpInterfaceDef.";
};

// Enumeration of valid operational interface roles.
enum OpInterfaceRole {ROLE_CLIENT, ROLE_SERVER, ROLE_CLIENT_SERVER};

// Class representing operational interface definitions.
class OpInterfaceDef : InterfaceDef {
    // The value for this attribute is derived from the causalities of the
    // operations contained in this interface.
    readonly derived attribute OpInterfaceRole role;

    // Operation to check if another operational interface is compatible (i.e.
    // bindable) with this one.
    boolean compatible(in string op_interface_id);

    OperationDef create_operation(in string id, in string name,
        in string version,
        in ::MetaORB::IDLType result,
        in OperationMode mode,
        in ParameterDescriptionSeq params,
        in ExceptionDefSeq exceptions,
        in stringSeq contexts);

    OperationDef create_prov_operation(
        in string id, in string name, in string version,
        in ::MetaORB::IDLType result,
        in OperationMode mode,
        in ParameterDescriptionSeq params,
        in ExceptionDefSeq exceptions,
        in stringSeq contexts,
        in ::MetaORB::QoSDefs::QoSAnnotationDef qos_annotation);

    OperationDef create_req_operation(
        in string id, in string name, in string version,
        in ::MetaORB::IDLType result,
        in OperationMode mode,
        in ParameterDescriptionSeq params,
        in ExceptionDefSeq exceptions,
        in stringSeq contexts,
        in ::MetaORB::QoSDefs::QoSAnnotationDef qos_annotation);

    ExceptionDef create_exception(in string id, in string name,
        in string version,
        in StructMemberSeq members);
};

typedef sequence<OpInterfaceDef> OpInterfaceDefSeq;

// Enumeration of the valid flow directions.
enum FlowDirection { FLOW_IN, FLOW_OUT };

// Class representing flow definitions.
class FlowDef : QoSConstrained, ::MetaORB::Contained {
    attribute FlowDirection direction;

    // Reference to the flow's media specification.
    reference media_spec to mediaSpec of FlowType;

    constraint FlowContainment "Eng" :
        "[C-12] A FlowDef can only be defined in a StrInterfaceDef.";

    constraint FlowTypeMustBeContinuous "Eng" :
        "[C-13] The media specification of a flow must be a continuous media
        type";
};

// Class representing stream interface definitions.
class StrInterfaceDef : InterfaceDef {
    // Operation to check if another stream interface is
    // compatible (i.e. bindable) with this one.
    boolean compatible(in string str_interface_id);

    FlowDef create_flow(in string id, in string name, in string version,
        in FlowDirection direction,
        in ::MetaORB::MediaDefs::MediaSpecificationDef media_spec,
        in ::MetaORB::QoSDefs::QoSAnnotationDef qos_annotation);
};

typedef sequence<StrInterfaceDef> StrInterfaceDefSeq;

// Class representing the description of signal values.
class ValueDescription : ::MetaORB::Typed {
    attribute string name;
}
Appendix B – Specifications of the Meta-ORB meta-model

```cpp
constraint SigValueMustNotBeContinuous "Eng" :
    "[C-21] The value of a signal must not be a continuous media type.;"
);
typedef sequence <ValueDescription> ValueDescriptionSeq;

// Enumeration of the valid signal directions.
enum SignalDirection { SIG_IN, SIG_OUT };)

// Class representing signal definitions.
class SignalDef : QoSConstrained, ::MetaORB::Contained {
    attribute SignalDirection direction;
    attribute ordered set [0..*] of ValueDescription values;
    constraint SignalContainment "Eng" :
        "[C-14] A SignalDef must be defined in a SigInterfaceDef;"
);

// Class representing signal interface definitions.
class SigInterfaceDef : InterfaceDef {
    // Operation to check if another signal interface is compatible (i.e.
    // bindable) with this one.
    boolean compatible(in string sig_interface_id);
    SignalDef create_signal(in string id, in string name, in string version,
        in SignalDirection direction,
        in ValueDescriptionSeq values,
        in ::MetaORB::QoSDefs::QoSAnnotationDef qos_annotation);
}
typedef sequence<SigInterfaceDef> SigInterfaceDefSeq;

// ** Associations **
association QoS {
    end set [0..*] of QoSConstrained qosConstrained;
    end set [0..1] of ::MetaORB::QoSDefs::QoSAnnotationDef qosAnnotation;
}

association BaseInterfaces {
    end set [0..*] of InterfaceDef derived_interf;
    end ordered set [0..*] of InterfaceDef base;
}

association CanRaise {
    end set [0..*] of OperationDef operationDef;
    end ordered set [0..*] of ExceptionDef exceptionDef;
}

// ** End of nested package Interfaces **

// ** Nested package **
package Components {
    // Class representing the base interface for component definitions.
    abstract class ComponentDef : ::MetaORB::Contained {
    };

    // Artefact class which serves as the base class for the model elements
    // representing the exposition of an existing interface definition with another
    // name or added properties.
    abstract class InterfaceExposer {
        attribute string interf_name;
        reference interf_type to interfType of CompInterfType;
    };

    // Class representing an interface of a primitive component.
    class PrimInterface : InterfaceExposer {
    };
typedef sequence <PrimInterface> PrimInterfaceSeq;
```
typedef sequence<octet> ImplemCode;

// Class representing primitive component definitions.
class PrimComponentDef : ComponentDef {
    attribute string implem_name;
    readonly attribute ImplemCode implem;
    reference interfaces to interf of PrimCompInterfs;

    // Operation to get one of the component's interfaces by its name.
    PrimInterface get_interface_by_name(in string interf_name);

    constraint PrimComponentContainment "Eng" :
        "[C-15] A PrimComponentDef can only be defined in a ModuleDef";
};

// Class representing an internal component of a composite one.
class InternalComponent {
    // The name of the internal component in the context of the composition.
    attribute string component_name;

    // Reference to the actual component type defining this internal component.
    reference component_type to compDef of IntCompType;
};
typedef sequence<InternalComponent> InternalComponentSeq;

// Structure representing a node in an object graph (or simply a pair
// component-interface).
struct GraphNode {
    string component_name;
    string interface_name;
};

// Structure representing a local binding (i.e. a pair of nodes in an object graph
// which happen to be linked together by their named interfaces). This structure
// may be augmented in future with further properties of local bindings.
struct LocalBindingDcl {
    GraphNode node_a;
    GraphNode node_b;
};
typedef sequence<LocalBindingDcl> LocalBindingDclSeq;

// Class representing an interface of a composite component.
class CompInterface : InterfaceExposer {
    // This attribute represents an interface of an internal component which is
    // exported as an external component interface.
    attribute GraphNode interfExpose;
};
typedef sequence<CompInterface> CompInterfaceSeq;

// Class representing composite component definitions.
class CompComponentDef : ComponentDef {
    attribute set [0..*] of LocalBindingDcl obj_graph;
    reference internal_comps to internalComp of InternalComps;
    reference interfaces to interf of CompInterfs;

    // Operation to get one of the component's interfaces by its name.
    CompInterface get_interface_by_name(in string interf_name);

    constraint CompComponentContainment "Eng" :
        "[C-16] A CompComponentDef can only be defined in a ModuleDef";
};

// ** Associations **
association CompInterfType {
    end set [0..*] of InterfaceExposer interf;
    end single ::MetaORB::Interfaces::InterfaceDef interfType;
};
association IntCompType {
    end set [0..*] of InternalComponent internalComp;
    end single ComponentDef compDef;
};
association PrimCompInterfs {
    composite end single PrimComponentDef compType;
}
Appendix B – Specifications of the Meta-ORB meta-model

end set [1..*] of PrimInterface interf;
};
association InternalComps {
    composite end single CompComponentDef compositeComp;
    end set [1..*] of InternalComponent internalComp;
};
association CompInterfs {
    composite end single CompComponentDef compType;
    end set [1..*] of CompInterface interf;
};
}; // ** End of nested package Components **

// ** Nested package **
package Bindings{

    // Enumeration of the valid matching rules for matching role
    // target interfaces to the actual interfaces to be bound.
    enum MatchingRule { STRICT, SUBTYPE, PARTIAL };

    // Structure representing the cardinality of a role.
    struct Cardinality{
        unsigned long min;
        unsigned long max;
    };

    // Structure representing the causal dependencies of a role.
    struct Dependencies{
        sequence <string> required;
        sequence <string> excluded;
    };

    // Class representing the base interface for role definitions.
    abstract class RoleDef : ::MetaORB::Contained {
        attribute MatchingRule matching_rule;
        attribute Cardinality cardinality;
        attribute Dependencies causal_dpend;
        attribute set [0..*] of string essential_interactions;

        // Operation to check if the target interface of this role
        // is satisfied by an actual interface to be bound.
        boolean satisfied_by(in ::MetaORB::Interfaces::InterfaceDef interf_def);

        constraint StrictMatchingRule "Eng" :
            "[C-17] If MatchingRule is PARTIAL, the attribute
            essential_interactions must not be left empty."
    };

    // Class representing a definition of a role for primitive bindings.
    class PrimRoleDef : RoleDef {
        reference interf_def to interfDef of PrimTargetInterf;

        constraint PrimRoleContainment "Eng" :
            "[C-18] A PrimRoleDef can only be defined in a PrimBindingDef."
    };

    // Class representing the target interface of roles of composite bindings.
    class TargetInterface : ::MetaORB::Components::InterfaceExposer {
        attribute ::MetaORB::Components::GraphNode interf_expose;
        reference interf_def to interfDef of InterfaceType;
    };

    // Class representing a definition of a role for composite bindings.
    class CompRoleDef : RoleDef {
        attribute set [0..*] of ::MetaORB::Components::LocalBindingDcl configuration;

        reference components to comp of InternalComps;
        reference target_interface to targetInterf of RoleTargetInterf;

        constraint CompRoleContainment "Eng" :
            "[C-19] A CompRoleDef can only be defined in a CompBindingDef."
    };

    // Class representing the base interface for binding definitions.
    abstract class BindingDef : ::MetaORB::Container {
constraint BindingContainment "Eng" :
  "[C-20] A BindingDef can only be defined in a ModuleDef.";
}.

typedef sequence<octet> ImplemCode;
typedef sequence<string> OptionalInteractions;

// Class representing primitive binding definitions.
class PrimBindingDef : BindingDef {
  attribute string implem_name;
  readonly attribute ImplemCode implem;

  PrimRoleDef create_role(in string id, in string name, in string version,
    in ::MetaORB::Interfaces::InterfaceDef target_interf,
    in MatchingRule matching_rule,
    in Cardinality cardinality,
    in Dependencies causal_depend,
    in OptionalInteractions essential_interactions);
};

// Class representing the control interface of composite bindings.
class ControlInterface {
  attribute string interf_name;

  // The type of the component that implements the control interface.
  attribute ::MetaORB::Components::ComponentDef implem_component;

  attribute string exposed_interface;

  constraint CtrlCompMustSupportInterface "Eng" :
    "[C-21] The type referred to by implem_component must support an
    interface of name exposed_interface.";
};
typedef sequence<ControlInterface> ControlInterfaceSeq;

// Class representing an internal binding in a composite binding.
class InternalBinding {
  attribute string binding_name;

  // Reference to the actual binding type of the internal binding.
  reference binding_type to bindingDef of IntBindingType;
};
typedef sequence<InternalBinding> InternalBindingSeq;

// Class representing composite binding definitions.
class CompBindingDef : BindingDef {
  reference control_interfaces to ctrlInterf of BindingCtrlInterfs;
  reference internal_bindings to intBinding of BindingContainment;

  RoleDef create_role(in string id, in string name, in string version,
    in ::MetaORB::Components::InternalComponentSeq components,
    in TargetInterface target_interf,
    in ::MetaORB::Components::LocalBindingDclSeq configuration,
    in MatchingRule matching_rule,
    in Cardinality cardinality,
    in Dependencies causal_depend,
    in OptionalInteractions essential_interactions);
};

// ** associations **
association IntBindingType {
  end set [0..*] of InternalBinding intBinding;
  end single BindingDef bindingDef;
};

association PrimTargetInterf {
  end set [0..*] of PrimRoleDef role;
  end single ::MetaORB::Interfaces::InterfaceDef interfDef;
};

association InterfaceType {
  end set [0..*] of TargetInterface targetInterf;
  end single ::MetaORB::Interfaces::InterfaceDef interfDef;
};

association RoleTargetInterf {
Appendix B – Specifications of the Meta-ORB meta-model

composite end single CompRoleDef role;
end single TargetInterface targetInterf;
}

association InternalComps {
    composite end single CompRoleDef role;
    end set [1..*] of ::MetaORB::Components::InternalComponent comp;
}

association BindingCtrlInterfs {
    composite end single CompBindingDef bindingType;
    end ordered set [1..*] of ControlInterface ctrlInterf;
}

association BindingContainment {
    composite end single CompBindingDef compBinding;
    end set [1..*] of InternalBinding intBinding;
}

}; // ** End of nested package Bindings **

// Class representing module model elements.
class ModuleDef : Container {
    // (These 'create' operations were originally defined in TopContainer. However,
    // to avoid circular dependencies between the packages, they now defined in
    // ModuleDef. This should not be a problem as such model elements can only be
    // created inside a module.)
    Interfaces::ExceptionDef create_exception(in string id, in string name,
        in string version,
        in StructMemberSeq members);
    Interfaces::OpInterfaceDef create_op_interface{
        in string id, in string name, in string version,
        in Interfaces::OpInterfaceDefSeq base_interfaces};
    Interfaces::StrInterfaceDef create_str_interface{
        in string id, in string name, in string version,
        in Interfaces::StrInterfaceDefSeq base_interfaces};
    Interfaces::SigInterfaceDef create_sig_interface{
        in string id, in string name, in string version,
        in Interfaces::SigInterfaceDefSeq base_interfaces};
    Components::PrimComponentDef create_prim_component{
        in string id, in string name, in string version,
        in string implem_name,
        in Components::PrimInterfaceSeq interfaces};
    Components::CompComponentDef create_component{
        in string id, in string name, in string version,
        in Components::InternalComponentSeq internal_components,
        in Components::LocalBindingDclSeq obj_graph,
        in Components::CompInterfaceSeq interfaces};
    Bindings::PrimBindingDef create_prim_binding(in string id, in string name,
        in string version,
        in string implem_name);
    Bindings::CompBindingDef create_biding{
        in string id, in string name, in string version,
        in Bindings::ControlInterfaceSeq control_interaces,
        in Bindings::InternalBindingSeq internal_bindings};
}

// Class representing the whole repository (i.e., the root container).
singleton class Repository : TopContainer{
    // Operation to lookup for a contained object based on its unique repository
    // identifier.
    Contained lookup_id(in string search_id);
    // Operation to obtain an anonymous primitive type (those referred to in the
    // PrimitiveKind enum).
    PrimitiveDef get_primitive(in PrimitiveKind kind);
    // Operations to create anonymous repository objects.
    StringDef create_string(in unsigned long bound);
Appendix B – Specifications of the Meta-ORB meta-model

WstringDef create_wstring(in unsigned long bound);
SequenceDef create_sequence(in unsigned long bound, in IDLType element_type);
ArrayDef create_array(in unsigned long length, in IDLType element_type);
FixedDef create_fixed(in unsigned short digits, in short scale);

// Operation to create a media type system object.
MediaDefs::MediaTypeSystemDef create_media_type_system(
in string id, in string name,
in string version);

// Operations to create QoS-related definitions.
QoSDefs::QoSBaseDef create_qos_attr_base(in string id, in string name,
in string version,
in QoSDefs::QoSCharacteristicSeq attr_descrs);

// (note that QoSAnnotationDefs are anonymous objects)
QoSDefs::QoSAnnotationDef create_qos_annotation(
in QoSDefs::QoSBaseDef qos_base,
in QoSDefs::QoSAttributeSeq attrs);

// Operation to create a new version of an existing type definition.
string new_version(in any type_definition);

};

}; // ** End of root package MetaORB **

B.3 ODL grammar specification

This section presents the complete specification of the Object Definition Language (ODL), which defines a concrete notation for specifying meta-information according to the Meta-ORB meta-model. The grammar is specified using BNF notation. Note that the productions that are identical to the standard OMG IDL specification are in normal font, while the ones corresponding to the Meta-ORB ODL extensions are shown in boldface. In addition, note that this specification is shown for informative purposes, as a compiler for the language has not been developed.

(1) <specification> ::= <definition> | <global_definition>
(2) <definition> ::= <type_dcl> ";"
    | <const_dcl> ";"
    | <except_dcl> ";"
    | <interface> ";"
    | <module> ";"
    | <media_spec> ";"
    | <component> ";"
    | <binding> ";"
(3) <module> ::= "module" <identifier> "{" <definition> "" }
(4) <interface> ::= <op_interface> | <stream_interface> | <signal_interface>
(5) <op_interface> ::= <interface_dcl>
    | <forward_dcl>
(6) <interface_dcl> ::= <interface_header> "{" <interface_body> ""}
(7) <forward_dcl> ::= "interface" ["<operational>"], <identifier>

(8) <interface_header> ::= "interface" ["<operational>"]
   <identifier> [<inheritance_spec>]

(9) <interface_body> ::= <export>*

(10) <export> ::= <type_dcl> ";"
    | <const_dcl> ";"
    | <except_dcl> ";"
    | <attrib_dcl> ";"
    | <media_spec> ";"
    | { <provided_ops> | <required_ops> } ";"

(11) <provided_ops> ::= ["provides:"] { <op_descr> ";" }*

(12) <required_ops> ::= "requires:" [ <op_descr> ";" ]*

(13) <op_desc> ::= <op_dcl> [<qos_annot_dcl>]

(14) <op_dcl> ::= [ <op_attribute> ] <op_type_spec> <identifier>
                   <parameter_dcls>
                   [ <raises_expr> ] [ <context_expr> ]

(15) <op_attribute> ::= "oneway"

(16) <op_type_spec> ::= <param_type_spec> | "void"

(17) <parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> "")"
    | "(" ""

(18) <param_dcl> ::= <param_attribute> <param_type_spec> <simple_declarator>

(19) <param_attribute> ::= "in" | "out" | "inout"

(20) <param_type_spec> ::= <base_type_spec>
    | <string_type>
    | <wide_string_type>
    | <fixed_pt_type>
    | <scoped_name>
    | <discrete_media_type>

(21) <discrete_media_type> ::= "media" <media_spec_name>

(22) <media_spec_name> ::= <scoped_name>

(23) <raises_expr> ::= "raises" ["<scoped_name>
    { "," <scoped_name> "")"

(24) <context_expr> ::= "context" ["<string_literal>
    { "," <string_literal> "")"

(25) <except_dcl> ::= "exception" <identifier> ["<member>" ""

(26) <stream_interface> ::= <stream_interface_dcl>
    | <stream_forward_dcl>

(27) <stream_interface_dcl> ::= <stream_interface_header>
    ["<stream_interface_body> ""]
(28) `<stream_forward_dcl> ::= "interface" "<stream>" <identifier>`

(29) `<stream_interface_header> ::= "interface" "<stream>"
    <identifier> [inheritance_spec]`

(30) `<stream_interface_body> ::= <stream_export>*`

(31) `<stream_export> ::= <type_dcl> ";" 
    | <const_dcl> ";"
    | <attr_dcl> ";"
    | <media_spec> ";"
    | <flow_dcl> ";"

(32) `<flow_dcl> ::= <flow_direction>
    <flow_name>
    "(" <flow_type> ")"
    [qos_annotation_dcl]`

(33) `<flow_direction> ::= "flowIn" | "flowOut"

(34) `<flow_name> ::= <identifier>`

(35) `<flow_type> ::= <media_spec_name>`

(36) `<signal_interface> ::= <signal_interface_dcl> | <signal_forward_dcl>`

(37) `<signal_interface_dcl> ::= <signal_interface_header>
    "{" <signal_interface_body> "}"`

(38) `<signal_forward_dcl> ::= "interface" "<signal>" <identifier>`

(39) `<signal_interface_header> ::= "interface" "<signal>"
    <identifier> [inheritance_spec]`

(40) `<signal_interface_body> ::= <signal_export>*`

(41) `<signal_export> ::= <type_dcl> ";" 
    | <const_dcl> ";"
    | <attr_dcl> ";"
    | <media_spec> ";"
    | <signal_dcl> ";"

(42) `<signal_dcl> ::= <signal_direction> <signal_name> "(" [signal_values] ")"
    [qos_annotation_dcl]`

(43) `<signal_direction> ::= "sigIn" | "sigOut"

(44) `<signal_name> ::= <identifier>`

(45) `<signal_values> ::= <sig_value_dcl> "{"," sig_value_dcl}"`

(46) `<sig_value_dcl> ::= <sig_value_type> <sig_value_name>`

(47) `<sig_value_type> ::= <param_type_spec>`

(48) `<sig_value_name> ::= <identifier>`
Appendix B – Specifications of the Meta-ORB meta-model

(49) `<attr_dcl>` ::= [ "readonly" ] "attribute"
    `param_type_spec` `<simple_declarator>`
    `;` `<simple_declarator>` `)`

(50) `<scoped_name>` ::= `<identifier>`
    | `::` `<identifier>`
    | `<scoped_name>` `::` `<identifier>`

(51) `<const_dcl>` ::= `<const_dcl>` `::="const"` `<const_type>`
    `<identifier>` `="` `<const_exp>`

(52) `<const_type>` ::= `<integer_type>`
    | `<char_type>`
    | `<boolean_type>`
    | `<floating_pt_type>`
    | `<string_type>`
    | `<wide_string_type>`
    | `<fixed_pt_const_type>`
    | `<scoped_name>`
    | `<octet_type>`

(53) `<const_exp>` ::= `<or_expr>`

(54) `<or_expr>` ::= `<xor_expr>` | `<or_expr>` `|` `<xor_expr>`

(55) `<xor_expr>` ::= `<and_expr>`
    | `<xor_expr>` `^` `<and_expr>`

(56) `<and_expr>` ::= `<shift_expr>`
    | `<and_expr>` `&` `<shift_expr>`

(57) `<shift_expr>` ::= `<add_expr>`
    | `<shift_expr>` `>>` `<add_expr>`
    | `<shift_expr>` `<<` `<add_expr>`

(58) `<add_expr>` ::= `<mult_expr>`
    | `<add_expr>` `+` `<mult_expr>`
    | `<add_expr>` `-` `<mult_expr>`

(59) `<mult_expr>` ::= `<unary_expr>`
    | `<mult_expr>` `*` `<unary_expr>`
    | `<mult_expr>` `/` `<unary_expr>`
    | `<mult_expr>` `%` `<unary_expr>`

(60) `<unary_expr>` ::= `<unary_operator>` `<primary_expr>`
    | `<primary_expr>`

(61) `<unary_operator>` ::= `~` | `+` | `-`

(62) `<primary_expr>` ::= `<scoped_name>`
    | `<literal>`
    | `("` `<const_exp>` `")`

(63) `<literal>` ::= `<integer_literal>`
    | `<string_literal>`
    | `<wide_string_literal>`
    | `<character_literal>`
    | `<wide_character_literal>`
(64) <boolean_literal> ::= “TRUE” | “FALSE”

(65) <positive_int_const> ::= <const_expr>

(66) <type_dcl> ::= “typedef” <type_declarator>
                | <struct_type>
                | <union_type>
                | <enum_type>
                | “native” <simple_declarator>

(67) <type_declarator> ::= <type_spec> <declarators>

(68) <type_spec> ::= <simple_type_spec> | <constr_type_spec>

(69) <simple_type_spec> ::= <base_type_spec>
           | <template_type_spec>
           | <scoped_name>

(70) <base_type_spec> ::= <floating_pt_type>
           | <integer_type>
           | <char_type>
           | <wide_char_type>
           | <boolean_type>
           | <octet_type>
           | <any_type>

(71) <template_type_spec> ::= <sequence_type>
           | <string_type>
           | <wide_string_type>
           | <fixed_pt_type>

(72) <constr_type_spec> ::= <struct_type> | <union_type> | <enum_type>

(73) <declarators> ::= <declarator> { “,” <declarator> }*

(74) <declarator> ::= <simple_declarator> | <complex_declarator>

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

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

(77) <floating_pt_type> ::= “float”
            | "double"
            | "long" "double"

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

(79) <signed_int> ::= <signed_short_int>
            | <signed_long_int>
            | <signed_longlong_int>

(80) <signed_short_int> ::= “short”

(81) <signed_long_int> ::= “long”
Appendix B – Specifications of the Meta-ORB meta-model

(82) `<signed_longlong_int>` ::= “long” “long”

(83) `<unsigned_int>` ::= `<unsigned_short_int>`
    | `<unsigned_long_int>`
    | `<unsigned_longlong_int>`

(84) `<unsigned_short_int>` ::= “unsigned” “short”

(85) `<unsigned_long_int>` ::= “unsigned” “long”

(86) `<unsigned_longlong_int>` ::= “unsigned” “long” “long”

(87) `<char_type>` ::= “char”

(88) `<wchar_type>` ::= “wchar”

(89) `<boolean_type>` ::= “boolean”

(90) `<octet_type>` ::= “octet”

(91) `<any_type>` ::= “any”

(92) `<struct_type>` ::= “struct” `<identifier>` “{” `<member_list> “}”

(93) `<member_list>` ::= `<member>`

(94) `<member>` ::= `<type_spec>` `<declarators>` “;”

(95) `<union_type>` ::= “union” `<identifier>` “switch”
    “{” `<switch_type_spec> “}”
    “{” `<switch_body> “}”

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

(97) `<switch_body>` ::= `<cases>`

(98) `<cases>` ::= `<case_label>` `<element_spec>` “;”

(99) `<case_label>` ::= “case” `<const_expr> “;”
    | “default” “;”

(100) `<element_spec>` ::= `<type_spec>` `<declarator>

(101) `<enum_type>` ::= “enum” `<identifier>`
    “{” `<enumerator> { “,” `<enumerator> } ”}”

(102) `<enumerator>` ::= `<identifier`

(103) `<sequence_type>` ::= “sequence” “<” `<simple_type_spec> “,”
    `<positive_int_const> “>”
    | “sequence” “<” `<simple_type_spec> “>”

(104) `<string_type>` ::= “string” “<” `<positive_int_const> “>”
    | “string”

(105) `<wide_string_type>` ::= “wstring” “<” `<positive_int_const> “>”
Appendix B – Specifications of the Meta-ORB meta-model

(106) <array_declarator> ::= <identifier> <fixed_array_size>

(107) <fixed_array_size> ::= [" <positive_int_const> "]

(108) <fixed_pt_type> ::= "fixed" <positive_int_const> ”,” <positive_int_const> ”>”

(109) <fixed_pt_const_type> ::= "fixed"

(110) <qos_annotation_dcl> ::= “QoS” "{ <qos_attr_dcl> { “,” <qos_attr_dcl> }* }"

(111) <qos_attr_dcl> ::= <qos_attr_tag> "=” <qos_attr_value>

(112) <qos_attr_tag> ::= <string_literal>

(113) <qos_attr_value> ::= <simple_value>
    | <range_value>

(114) <simple_value> ::= <floating_pt_literal>

(115) <range_value> ::= <integer_literal> ".." <integer_literal>
    | <fixed_literal> ".." <fixed_literal>
    | <floating_pt_literal> ".." <floating_pt_literal>

(116) <media_spec> ::= "media" "type" <identifier> ";" <major_media_type>
        {"<specific_minor_media_type_list> "}" ;"

(117) <specific_minor_media_type_list> ::= "encoding"
        "<" <minor_media_type_name_list> "" {"<media_attr_list> "}" 
        { "encoding"
        "<" <minor_media_type_name_list> "" 
        {"<media_attr_list> "}"
        }

(118) <minor_media_type_name_list> ::= <minor_media_type_name>
        { ";" <minor_media_type_name> }

(119) <minor_media_type_name> ::= <scoped_name>

(120) <media_attr_list> ::= NULL
    | <media_attr> { ",” <media_attr> }

(121) <media_attr> ::= <media_attr_name> "=" <media_attr_value>

(122) <media_attr_name> ::= <string_literal>

(123) <media_attr_value> ::= <simple_value> | <multiple_values>

(124) <simple_value> ::= <literal>

(125) <multiple_values> ::= <alternative_values>
    [ <range_value> "::" ] [ <range_qualifier> ]

(126) <alternative_values> ::= "<" <single_value_list> ">"
(127) <single_value_list> ::= <single_value> { ‘,”<single_value> }*

(128) <range_qualifier> ::= “minimise” | “maximise”

(129) <major_media_type> ::= <continuous_major_media_type>
    | <discrete_major_media_type>

(130) <continuous_major_media_type> ::= “audio” | “video” | “animation”

(131) <discrete_major_media_type> ::= “image” | “text”

(132) <component> ::= <prim_component> | <comp_component>

(133) <prim_component> ::= <prim_component_header>
    “{“<prim_component_body> ‘}” ;

(133) <prim_component_header> ::= “primitive” “component” <identifier>

(134) <prim_component_body> ::= <implem_dcl> <prim_interf_dcl>

(135) <implem_dcl> ::= “implementation” “;” <string_literal> “;”

(135) <prim_interf_dcl> ::= “interfaces” “;” <prim_interf> *

(136) <prim_interf> ::= <interf_type_name> <identifier> “;”

(137) <interf_type_name> ::= <scoped_name>

(138) <comp_component> ::= <comp_component_header>
    “{“<comp_component_body> ‘}” ;

(139) <comp_component_header> ::= “component” <identifier>

(140) <comp_component_body> ::= <internal_components_dcl>
    <graph_dcl>
    <comp_interf_dcl>

(141) <internal_components_dcl> ::= “internal” “components” “;”
    { <component_type_name>
      <component_names> “;” }

(142) <component_type_name> ::= <scoped_name>

(143) <component_names> ::= <identifier> { “,” <identifier> }*

(144) <graph_dcl> ::= “object” “graph” “;” <local_binding_dcl> *

(145) <local_binding_dcl> ::= <graph_node> “;” <graph_node> “;”

(146) <graph_node> ::= “{“<component_name> “;” <interf_name> ‘}”

(147) <component_name>, <interf_name> ::= <identifier>

(148) <comp_interf_dcl> ::= “interfaces” “;” <comp_interf> *

(149) <comp_interf> ::= <interf_type_name> <identifier> “is” <interf_expose> “;”

(150) <interf_expose> ::= <graph_node>
(151) <binding> ::= <prim_binding> | <comp_binding>

(152) <prim_binding> ::= <prim_binding_header> "{" <prim_binding_body> "}" ;

(153) <prim_binding_header> ::= "primitive" "binding" <identifier>

(154) <prim_binding_body> ::= <bind_impl_dcl> <prim_role_dcl> ;

(155) <bind_impl_dcl> ::= "Implementation" "string literal" ;

(156) <prim_role_dcl> ::= "role" <identifier> :
    "{" <prim_target_interf_dcl>
    [ <match_dcl> ]
    [ <card_dcl> ]
    [ <depend_dcl> ] "}" ;

(157) <prim_target_interf_dcl> ::= "target interface" "string literal" <interf_type_name> ;

(158) <comp_binding> ::= <comp_binding_header> "{" <comp_binding_body> "}" ;

(159) <comp_binding_header> ::= "binding" <identifier>

(160) <comp_binding_body> ::= <ctrl_interf_dcl>
    <internal_bindings_dcl>
    <comp_role_dcl> ;

(170) <ctrl_interf_dcl> ::= "control" "interfaces" ;
    { <interf_type_name>
    <interf_name> "is"
    "string literal" <interf_implem> "}" ;

(171) <interf_type_name> ::= <scoped_name>

(172) <interf_name> ::= <identifier>

(173) <interf_implem> ::= <component_type_name> "," <exposed_interf_name>

(174) <exposed_interf_name> ::= <identifier>

(175) <internal_bindings_dcl> ::= "internal" "bindings" ;
    { <binding_type> <binding_name> ; } ;

(176) <binding_type> ::= <scoped_name>

(177) <binding_name> ::= <identifier>

(178) <comp_role_dcl> ::= "role" <identifier> :
    "{" <compone_nts_dcl>
    <target_interf_dcl>
    [ <match_dcl> ]
    [ <card_dcl> ]
    [ <depend_dcl> ]
    <config_dcl> "}" ;

(179) <components_dcl> ::= "components" ;
    { <component_type_name> <component_names> ";" } ;
Appendix B – Specifications of the Meta-ORB meta-model

(180) \( <\text{component\_type\_name}> ::= <\text{scoped\_name}> \)

(181) \( <\text{component\_names}> ::= <\text{component\_name}> \{ \text{"","} <\text{component\_name}> \} * \)

(182) \( <\text{component\_name}> ::= <\text{identifier}> \)

(183) \( <\text{target\_interf\_dcl}> ::= \text{"target interface" }\)"
   \( <\text{interf\_type\_name}> <\text{interf\_expose}> \";\"

(184) \( <\text{interf\_expose}> ::= \text{"is" }<\text{graph\_node}> \)

(185) \( <\text{config\_dcl}> ::= \text{"configuration" }\)"
   \( <\text{role\_local\_binding}> \)

(186) \( <\text{role\_local\_binding}> ::= <\text{role\_graph\_node}> \";\"<\text{role\_graph\_node}> \";\"

(187) \( <\text{role\_graph\_node}> ::= \text{"(<\text{object\_name}> \",\" <\text{interact\_point\_name}> \)}"

(188) \( <\text{object\_name}> ::= <\text{binding\_name}> | <\text{component\_name}> \)

(189) \( <\text{interact\_point\_name}> ::= <\text{interf\_name}> | <\text{role\_name}> \)

(190) \( <\text{binding\_name}> , <\text{component\_name}> , <\text{interf\_name}> , <\text{role\_name}> ::= <\text{identifier}> \)

(191) \( <\text{match\_dcl}> ::= \text{"matching" }\)"
   \( \{ \text{"\text{STRICT}" | \"\text{SUBTYPE}" | <\text{partial\_match}> } \} \";\"

(192) \( <\text{partial\_match}> ::= \text{"\text{PARTIAL}" }\text{"(<\text{essential\_interactions}> \)}"

(193) \( <\text{essential\_interactions}> ::= \text{\text{NULL}}
   \text{ | <\text{identifier}> \{ \";\" <\text{identifier}> \} } * \)

(194) \( <\text{card\_dcl}> ::= \text{"cardinality" }\)"
   \( \{ <\text{integer\_literal}> | <\text{integer\_literal}> \"..\" <\text{integer\_literal}> \} \";\"

(195) \( <\text{depend\_dcl}> ::= \text{"dependencies" }\)"
\( [ <\text{req\_dep}> ] [ <\text{excl\_dep}> ] \)

(196) \( <\text{req\_dep}> ::= \text{"requires" }<\text{role\_name\_list}> \";\"

(197) \( <\text{excl\_dep}> ::= \text{"excludes" }<\text{role\_name\_list}> \";\"

(198) \( <\text{role\_name\_list}> ::= <\text{role\_name}> \{ \";\" <\text{role\_name}> \} * \)

(199) \( <\text{role\_name}> ::= <\text{identifier}> \)

(200) \( <\text{global\_definition}> ::= <\text{qos\_base\_dcl}> | <\text{media\_type\_system}> \)

(201) \( <\text{qos\_base\_dcl}> ::= \"\text{QoS base}" \{ <\text{qos\_charact\_dcl}>\"\} \)

(202) \( <\text{qos\_charact\_dcl}> ::= <\text{identifier}> \";\" <\text{value\_kind}> \";\" <\text{unit}> \";\"

(203) \( <\text{value\_kind}> ::= \text{TARGET} | \text{MAX} | \text{MIN} | \text{RANGE\_MAX} | \text{RANGE\_MIN} \)

(204) \( <\text{unit}> ::= \text{ABS} | \text{PERCENT} | \text{MSEC} | \text{SEC} | \text{BYTE} | \text{KB} | \text{MB} | \text{BPS} | \text{KBPS} | \text{MBPS} \)

(205) \( <\text{media\_type\_system}> ::= \text{\text{"media""type""system" }<\text{identifier}> \{ <\text{generic\_minor\_media\_type\_dcl}>\"\} \)"
B.4 Examples of ODL definitions

Figure B.1 shows the definition of an operational interface for a hypothetical binding factory, along with the accessory type definitions. Note that the keywords "<operational>" and "provided" are defined as optional in the grammar, and, if omitted, the interface definition becomes standard OMG IDL code.

```
module Example {
    enum StatusCode {SUCCESS, FAILURE};
    struct IRef {
        string interf_type_id;
        string location;
    };
    typedef sequence <IRef> IRefList;
    interface <operational> BindingFactory {
        readonly attribute long factory_id;
        provided: 
            StatusCode bind(in IRefList ifrefs);
            StatusCode bind_type(in IRefList ifrefs,
                                   in string binding_type_id);
    };
}
```

Figure B.1 – Operational interface definition

Figure B.2 in turn presents the definition of a primitive component representing the hypothetical binding factory. For brevity, the second interface is not shown here.
module Example {
    primitive component BindingFactoryComp {
        implementation: BindingFactoryImplem;
        interfaces:
            BindingFactory BF;
            BindingFactoryCtrl BFCtrl;
    }
};

Figure B.2 – Primitive component definition

module Example {
    media type system SimpleMediaTypes {
        generic media AudioPCM (audio) {
            attributes:
                short precision = <8,16>;  
                long frequency = <8000, 11025, 22050, 44100>;  
                short channels = <1,2>;  
        }
        generic media u-Law (audio) {
            attributes:
                short precision = 8;
                long frequency = <8000, 11025, 22050, 44100>;
                short channels = <1,2>;  
        }
        generic media MJPEG (video) {
            attributes:
                short X_resolution = 320..640;
                short Y_resolution = 200..480;
                short frame_rate = 15..30;
                short depth = <8,16,32>;  
        }
    }
    QoS base {
        throughput: RANGE_MAX, MBPS;
        delay: TARGET, MSEC;
        jitter: MAX, PERCENT;
        error_rate: MAX, PERCENT;
    }
};

Figure B.3 – Media type system and QoS base definitions

An example with a media type system and a QoS base definition is presented in Figure B.3 above. The media type system defines two generic (minor) media types, one for PCM audio and the other for motion JPEG video (the names between brackets identify the major media types). In each case, the attributes of the generic media type are specified in terms of their data types, their names, and their valid sets of values, which in turn are specified either as alternative values (values between angled brackets) or as a range of values (two values separated by "..."). Importantly, in the case of alternative values, their order of preference for media type negotiation is from left to right, while in the case of value ranges, a preference for maximisation or minimisation should be specified in the media type specifications. The QoS base in turn defines four QoS characteristics that can be used to check the definitions of QoS annotations. Each QoS characteristic is specified by its name, followed by a declaration of the kind of value (e.g. target value, maximum value, and range of...
values with priority for maximisation) and the unit of measure used (e.g., millisecond, megabits per second, and percentage). Note that a data type is not specified for QoS characteristics, as in the current definition of the meta-model their values can only be of a numeric type (double).

```
module Example {
    media type SimpleAudio: audio {
        encoding <PCM> () | encoding <u-Law> ()
    }
    interface <stream> AudioDevice {
        flowOut audio_out (SimpleAudio)
        QoS(delay = 100, jitter = 5, error_rate = 0);
        flowIn audio_in (SimpleAudio)
        QoS(delay = 150, jitter = 5, error_rate = 0.1);
    }
    media type SimpleVideo: video {
        encoding <MJPEG> (X_resolution = maximise,
        Y_resolution = maximise,
        frame_rate = 25..30 : maximise)
    }
    interface <stream> VideoDevice {
        flowOut video_out (SimpleVideo)
        QoS(delay = 100, jitter = 10, error_rate = 0.5);
        flowIn video_in (SimpleVideo)
        QoS(delay = 200, jitter = 10, error_rate = 1);
    }
}
```

**Figure B.4 – Stream interface and media specification definitions**

Two stream interface definitions are then presented in Figure B.4, representing two hypothetical media devices for audio and video, respectively. Each interface has two flows, one for producing and the other for consuming a media stream. The media types for the flows are defined in the two media type specifications. The first one, SimpleAudio, specifies two alternative audio encoding formats, based on the generic media types PCM and u-Law, which were defined above. The absence of attributes for each of the encoding specifications means that the attributes are taken directly (without any refinement) from the generic media type definitions. The second media type specification in turn specifies a single encoding for video, with attributes that refine those defined in the corresponding generic media type (MJPEG). The first two attributes are refined with the "maximise" directive, which instructs a QoS negotiation protocol to maximise the agreed value for the attributes. The third attribute then refines the value defined in the generic media type by narrowing the range and specifying the "maximise" directive (note that the attribute depth is "inherited" directly from the generic media type). Finally, note the QoS annotations associated with the flows, each one specifying a set of attribute-value pairs, which must conform
to the QoS characteristics defined in the QoS attribute base. Importantly, each attribute is interpreted according to the definition of the respective QoS characteristic.

A composite component definition is presented in Figure B.5, along with the supporting definitions. The composite represents a hypothetical audio-video device, capable producing and consuming audio and video streams. It is composed of three primitive internal components, linked according to the specified object graph (audio_comp and mixer_comp are linked by their interfaces named audio_interf, and video_comp is linked to mixer_comp via their interfaces named video_interf). In addition, the composite component has an interface of type AVDevice, which is the interface av_interf exported by the internal component mixer_comp. Note that the type of this interface is defined by inheriting from the already defined interface types AudioDevice and VideoDevice.

```plaintext
module Example {
  primitive component AudioDeviceComp {
    implementation: AudioDeviceImpl;
    interfaces: AudioDevice audio_interf;
  };
  primitive component VideoDeviceComp {
    implementation: VideoDeviceImpl;
    interfaces: VideoDevice video_interf;
  };
  interface <stream> AVDevice : AudioDevice, VideoDevice {};
  primitive component MixerComp {
    implementation: MixerImpl;
    interfaces: AudioDevice audio_interf;
    VideoDevice video_interf;
    AVDevice av_interf;
  };
  component AVDeviceComp {
    internal components: AudioDeviceComp audio_comp;
    VideoDeviceComp video_comp;
    MixerComp mixer_comp;
    object graph: (audio_comp, audio_interf): (mixer_comp, audio_interf);
    (video_comp, video_interf): (mixer_comp, video_interf);
    interfaces: AVDevice av is (mixer_comp, av_interf);
  };
}
```

Figure B.5 – Composite component definition

As a last example, Figure B.6 shows the definition of a hypothetical composite binding type, meant to define the connection between the interfaces of components of type AVDeviceComp defined above. The supporting definitions are omitted for brevity. The binding has a single control interface, named ctrl, which is supposed to have operations for starting, pausing and destroying the binding (the type of this interface, as well as its supporting component definition are not shown). The internal configuration of the binding is composed by two nested bindings, in addition to a few components (which are specified in the role definition as described below). The types
of the two nested bindings are referenced and names are given to the respective bindings in the context of the composite binding. The types of the two nested bindings are not shown, as their definitions are similar to the definition of the composite. The configuration of the binding is then specified in terms of its role definitions. In this case, a single role is defined, as all binding endpoints are of the same kind. The role definition, named `AVBindingPartic`, specifies the following features:

- the types and names of components that make up the configuration of the binding endpoints conforming to the role definition;

- the type of interface that an endpoint conforming to the role is expected to bind (as well as the mapping of an interface of an internal component, in order to provide the endpoint’s interface);

- the cardinality of the role, expressing a constraint on the number of endpoints conforming to the role that can exist in a binding of this type; in this case, there can be always exactly 2 endpoints conforming to the `AVBindingPartic` role; and

- the role configuration, which specifies how components and nested bindings are connected in order to support an endpoint of the binding; the whole binding configuration is assembled, when creating the binding, by combining the configurations of its several endpoints.

```plaintext
module Example {
    binding AVBinding {
        control interfaces: CtrlInterf ctrl is (CtrlComp, ctrl_interf);
        internal bindings: AudioBinding audio_binding;
                              VideoBinding video_binding;
        role AVBindingPartic {
            components: AVStubComp stub;
                        AudioFilterComp audio_filter;
                        VideoFilterComp video_filter;
            target interface: AVDevice is (stub, av_interf);
            cardinality: 2;
            configuration:
                (stub, audio_interf):(audio_filter, audio_interf);
                (stub, video_interf):(video_filter, video_interf);
                (audio_filter, forward_interf):(audio_binding, audio_role);
                (video_filter, forward_interf):(video_binding, video_role);
        }
    }
}
```

Figure B.6 – Binding definition
Appendix C  IDL / ODL Definitions of the Prototype

C.1 Introduction

This appendix accompanies Chapter 6 and provides the ODL definitions for the three modules described in that chapter, which constitute the Meta-ORB implementation, namely, the core facilities of the platform, the Type Repository, and the reflective meta-objects.

C.2 Core services interfaces

Overview

In this section, the complete definition of the types of the basic services of the platform are presented. Notably, the definition of the default component and binding factories are shown, along with their auxiliary definitions. The definitions are presented in ODL syntax (see Appendix A).

```
#include <CORBA.idl>

module ORBcore{
    exception ORBcoreException{
        string value;
    };

    /* Interface references */
    struct IRef {
        string interface_uname;
        CORBA::RepositoryId interface_type_id;
        string host;
        string capsule_id;
        unsigned long comm_server_port;
    };
    typedef sequence <IRef> IRefSeq;

    Configuration interface of binding control components

    struct NestedBindingInterf{
        string binding_name;
        IRef binding_ctrl_iref;
    };
    typedef sequence <NestedBindingInterf> NestedBindingInterfSeq;

    struct EndpointInfo{
        string capsule_uname;
        CORBA::Identifier role_name;
    };
    typedef sequence <EndpointInfo> EndpointList;

    interface BindingCtrlConfig{
```
// This meta-info is provided to the control component so that it is able to
// communicate with and control the internal bindings. It is not meant to be
// retrieved through this interface.

oneway void set_nested_bindings(in NestedBindingInterfSeq nested_ctrl_interfs);

oneway void set_endpoints(in EndpointList endpoint_list);
EndpointList get_endpoints();

oneway void set_binding_type_id(in CORBA::RepositoryId binding_type_id);

oneway void set_binding_name(in string binding_name);

Base type for binding control interfaces

Binding control interfaces are essentially binding-dependent features. Nevertheless,
a basic interface type is defined, which is highly recommended as a base type to be
inherited by the types of the main control interfaces of binding objects. This base
interface, defined below, defines two operations used to get the name and the type of
the binding. Examples of operations that can be added by a derived control interface
type include stop and start, which are useful when doing configuration adaptations.

interface BasicBindingCtrlInterf {
    string get_binding_name();
    CORBA::RepositoryId get_binding_type_id();
};

Binding control component

This component type is user-defined, and the only requirement is that it supports
the config_interf interface, of the type BindingCtrlConfig defined above,
plus a control interface of some appropriate type (derived from
BasicBindingCtrlInterf). Below is an example of a control component type.

primitive component BindingCtrlComp{
    implementation: BindingCtrlComp_impl;
    interfaces: BindingCtrlConfig config_interf;
    BindingCtrl ctrl_interf; //the type of this interface is user-defined
};

Configuration interface of primitive bindings

interface PBConfigInterf{
    oneway void set_endpoints(in EndpointList endpoint_list);
    EndpointList get_endpoints();
    oneway void set_binding_type_id(in CORBA::RepositoryId binding_type_id);
    CORBA::RepositoryId get_binding_type_id();
    oneway void set_binding_name(in string binding_name);
    string get_binding_name();
};

Configuration component for primitive bindings

primitive component PBConfigComp{
    implementation: PBConfigComp_impl;
    interfaces: PBConfigInterf config_interf;
};
Default Binding Factory interfaces

/* Primary interface */
interface BFInterf{
    BindingCtrl new(in IRefSeq iref_list,
                   in CORBA::RepositoryId binding_def_id,
                   in string binding_name)
        raises (ORBcoreException);
}

/* Secondary interface (and auxiliary definitions) */
typedef sequence <octet> ProtData;

struct PrimBindingResult {
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    ProtData finalProtocolData;
};

struct BindingResult {
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    struct NestedBindingResult{
        sequence <BindingResult> bindingResults;
        sequence <PrimBindingResult> primBindingResults;
    } nestedResults;
};

interface BFsecInterf{
    oneway void new_sec(in IRef target_iref,
                         in CORBA::RepositoryId binding_def_id,
                         in CORBA::RepositoryId role_def_id,
                         in string binding_name,
                         in unsigned long token,
                         in IRef init_iref);

    oneway void activate_endpoint(in BindingResult binding_result);

    oneway void cancel_endpoint(in string binding_name,
                                in string role_name);
}

/* Collect interface (and auxiliary definitions) */
enum SecStatusValues {SEC_NORMAL, SEC_FAIL_COMPONENT, SEC_FAIL_RESOURCE,
                      SEC_FAIL_NESTED, SEC_FAIL_ARGUMENTS, SEC_FAIL_BAD_DEF};

enum PrimSecStatusValues {PRIM_NORMAL, PRIM_FAIL_ARGUMENTS, PRIM_FAIL_IMPL};

struct PrimSecResult{
    PrimSecStatusValues status;
    string failureReason;
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    string roleName;
    ProtData protocolData;
};

struct SecResult{
    SecStatusValues status;
    string failureReason;
    string bindingName;
    CORBA::RepositoryId bindingDefId;
    string roleName;
    string capsuleName;
    struct EncapsResult{
        sequence <SecResult> secResult;
        sequence <PrimSecResult> primSecResult;
    } encapsulatedResult;
    unsigned long token;
};

interface BFcollectInterf{
    oneway void collect_sec_result(SecResult result);
}
Appendix C – IDL / ODL Definitions of the Prototype

**Default Binding Factory component**

```idl
primitive component BFcomp{
    implementation: BindingFactory;
    interfaces: BFInterf BF;
    BFsecInterf BFsec;
    BFcollectInterf BFcollect;
};
```

**Default Component Factory interface**

```idl
typedef sequence<any> ArgList;

interface CFInterf {
    string new(in CORBA::RepositoryId component_def_id,
                    in string component_name,
                    in string context_uname,
                    in boolean fixed_name,
                    in ArgList implem_args)
        raises (ORBcoreException);

    void destroy(in string component_uname)
        raises (ORBcoreException);
};
```

**Default Component Factory component**

```idl
primitive component CFcomp{
    implementation: ComponentFactory;
    interfaces: CFinterf CF;
};
```

**Binder interface and component**

This interface (and respective component) is meant to provide a *remote local binding service*, which allows entities in a remote location to request a local binding between two interfaces in the current capsule. (This is particularly useful for Architecture meta-objects that reify distributed binding objects).

```idl
interface BinderInterf {
    void local_bind(in string interf_uname1, in string interf_uname2)
        raises (ORBcoreException);

    void break_local_binding(in string interf_uname1, in string interf_uname2)
        raises (ORBcoreException);

    boolean test_role_interf(in string role_uname,
                              in CORBA::RepositoryId interf_type_id)
        raises (ORBcoreException);
};

primitive component BinderComp {
    implementation: BinderComp_impl;
    interfaces: BinderInterf Binder;
};
```

}; /* End of ORBCore module */
C.3 Type Repository Interfaces

Overview

This section presents the interfaces of the type repository objects. However, only
the interfaces that correspond to the meta-types newly defined by the Meta-ORB
meta-model are included. The remaining interfaces are as defined in the CORBA 2.2
specification [OMG 1998] (excluding the definitions of the interfaces InterfaceDef
and OperationDef, which are re-defined here), and are present in the included
module CORBA (despite the name, this module only contains features strictly related to
the Interface Repository definitions). In addition, note that, for brevity, the definitions
for TypeCodes are not shown here (they are based on the type code definitions
distributed as part of Fnorb [Fnorb 2000], with extensions corresponding to the Meta-
ORB meta-model).

The interface definitions are presented in ODL syntax, though only using standard
OMG IDL 2.2 features. The interfaces shown, as well as other data structures, are
mapped from the elements with the same name from the Meta-ORB meta-model (see
Chapter 4 and Appendix B).

QoS attributes and annotations

```idl
#include <CORBA.idl>

module MetaORB{

    enum QoSAttrValueKind {QOS_TARGET, QOS_RANGE_MAX, QOS_RANGE_MIN,
                           QOS_MAX, QOS_MIN };

    enum MeasureUnits { UNIT_ABS, UNIT_PERCENT, UNIT_MSEC, UNIT_SEC,
                        UNIT_B, UNIT_KB, UNIT_MB, UNIT_BPS,
                        UNIT_KBPS, UNIT_MBPS };

    struct QoSCharacteristic{
        CORBA::Identifier name;
        QoSAttrValueKind value_kind;
        MeasureUnits unit;
    };

typedef sequence<QoSCharacteristic> QoSCharacteristicSeq;

interface QoSBaseDef: CORBA::Contained{
    attribute QoSCharacteristicSeq attr_descrs;
};

struct QoSBaseDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSepc version;
};

struct QoSAttribute{
    CORBA::Identifier name;
    any value_min;
    any value_max;
}
```
typedef sequence <QoSAttribute> QoSAttributeSeq;

interface QoSAnnotationDef:IRObject{
    readonly attribute QoSAttrBaseDef qos_base;
    attribute QoSAttributeSeq attrs;

    QoSAnnotationDescription describe();
};

struct QoSAnnotationDescription{
    CORBA::RepositoryId qos_base;
    QoSAttributeSeq attrs;
};

Media types

enum MajorMediaTypes {MEDIA_ALL, MEDIA_AUDIO, MEDIA_VIDEO,
    MEDIA_ANIMATION, MEDIA_IMAGE, MEDIA_TEXT};

typedef sequence <any> MediaAttrValue;
enum MediaAttrValueKind {VALUE_SINGLE, VALUE_RANGE, VALUE_SEQUENCE};

struct GenericMediaAttrDescription{
    CORBA::Identifier name;
    TypeCode basic_type;
    CORBA::IDLType basic_type_def;
    MediaAttrValueKind value_kind;
    MediaAttrValue values;
};

typedef sequence <GenericMediaAttrDescription> GenericMediaAttrList;

interface GenericMediaTypeDef:Contained{
    readonly attribute MajorMediaTypes major_media_type;
    attribute GenericMediaAttrList attrs;

    SpecificMediaTypeDef specialise(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in CORBA::Container defined_in,
        in MajorMediaTypes major_media_type,
        in SpecificMediaAttrList spec_attr_list) raises (TRException);
};

struct GenericMediaTypeDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    MajorMediaTypes major_media_type;
    GenericMediaAttrList attrs;
};

interface MediaTypeSystemDef: Container, Contained{
    GenericMediaTypeDef create_generic_media_type(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in MajorMediaTypes major_media_type,
        in GenericMediaAttrList attrs);
};

struct MediaTypeSystemDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
};

typedef sequence <any> MediaAttrValue;

enum QualifierMode {QUALIF_NONE, QUALIF_MAXIMISE, QUALIF_MINIMISE};

struct SpecificMediaAttrDescription{
Appendix C – IDL / ODL Definitions of the Prototype

```idl
CORBA::Identifier name;
MediaAttrValueKind value_kind;
QualifierMode qualifier;
MediaAttrValue values;
}

typedef sequence <SpecificMediaAttrDescription> SpecificMediaAttrList;

interface SpecificMediaTypeDef: Contained{
    readonly attribute MajorMediaTypes major_media_type;
    readonly attribute GenericMediaTypeDef encoding;
    attribute SpecificMediaAttrList attrs;
}

struct SpecificMediaTypeDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    MajorMediaTypes major_media_type;
    CORBA::Identifier encoding;
    SpecificMediaAttrList attrs;
}

interface MediaSpecificationDef: TypedefDef, Container{
    readonly attribute MajorMediaTypes major_media_type;
    SpecificMediaTypeDef create_specific_media_type(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in MajorMediaTypes major_media_type,
        in CORBA::Identifier encoding,
        in SpecificMediaAttrList attrs);
}

Base interface for interface objects

typedef sequence <RepositoryId> RepositoryIdSeq;
typedef sequence <AttributeDescription> AttrDescriptionSeq;

interface InterfaceDef: CORBA::Container, CORBA::Contained, CORBA::IDLType {
    readonly attribute boolean is_abstract;
    boolean is_a(in CORBA::RepositoryId interface_id);
    boolean compatible(in CORBA::RepositoryId interface_id);
    CORBA::AttributeDef create_attribute{
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in CORBA::IDLType type,
        in CORBA::AttributeMode mode);
    MediaSpecificationDef create_media_specification{
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in string major_media_type);
}

enum InterfaceStyle {STYLE_OPERATIONAL, STYLE_STREAM, STYLE_SIGNAL};

struct InterfaceDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    RepositoryIdSeq base_interfaces;
    boolean is_abstract;
    InterfaceStyle style;
};
```
Appendix C – IDL / ODL Definitions of the Prototype

Operations

```cpp
struct ParameterDescription {  
    CORBA::Identifier name;  
    TypeCode type;  
    CORBA::ParameterMode mode;  
};
typedef sequence <ParameterDescription> ParDescriptionSeq;
enum OperationCausality {OP_PROVIDED, OP_REQUIRED};
interface OperationDef: CORBA::OperationDef{  
    readonly attribute boolean qos_constrained;  
    attribute QoSAnnotationDef qos_annotation;  
    attribute OperationCausality causality;  
};
struct OperationDescription{  
    CORBA::Identifier name;  
    CORBA::RepositoryId id;  
    CORBA::RepositoryId defined_in;  
    CORBA::VersionSpec version;  
    TypeCode result;  
    CORBA::OperationMode mode;  
    CORBA::ContextIdSeq contexts;  
    ParDescriptionSeq parameters;  
    CORBA::ExcDescriptionSeq exceptions;  
    boolean qos_constrained;  
    QoSAnnotationDescription qos_annotation;  
    OperationCausality causality;  
};
```

Operational interfaces

```cpp
interface OpInterfaceDef;  
typedef sequence <OpInterfaceDef> OpInterfaceDefSeq;  
enum OpInterfaceRole {ROLE_CLIENT, ROLE_SERVER, ROLE_CLIENT_SERVER};
interface OpInterfaceDef: InterfaceDef{  
    readonly attribute OpInterfaceRole role;  
    attribute OpInterfaceDefSeq base_interfaces;  
    struct FullOpInterfaceDescription{  
        CORBA::Identifier name;  
        CORBA::RepositoryId id;  
        CORBA::RepositoryId defined_in;  
        CORBA::VersionSpec version;  
        OpDescriptionSeq prov_operations;  
        OpDescriptionSeq req_operations;  
        CORBA::AttrDescriptionSeq attributes;  
        CORBA::RepositoryIdSeq base_interfaces;  
        boolean is_abstract;  
        TypeCode type;  
    };  
    FullOpInterfaceDescription describe_interface();  
    OperationDef create_operation(  
        in CORBA::RepositoryId id,  
        in CORBA::Identifier name,  
        in CORBA::VersionSpec version,  
        in CORBA::IDLType result,  
        in CORBA::OperationMode mode,  
        in CORBA::ParDescriptionSeq params,  
        in CORBA::ExceptionDefSeq exceptions,  
        in CORBA::ContextIdSeq contexts);  
    OperationDef create_prov_operation(  
        in CORBA::RepositoryId id,  
        in CORBA::Identifier name,  
        in CORBA::VersionSpec version,  
        in CORBA::IDLType result,  
        in CORBA::OperationMode mode,  
        in CORBA::ParDescriptionSeq params,
Appendix C – IDL / ODL Definitions of the Prototype

```
in CORBA::ExceptionDefSeq exceptions,
in CORBA::ContextIdsSeq contexts,
in QoSAnnotation qos_annotation);

OperationDef create_req_operation(
in CORBA::RepositoryId id,
in CORBA::Identifier name,
in CORBA::VersionSpec version,
in CORBA::IDLType result,
in CORBA::OperationMode mode,
in CORBA::ParamDescriptionSeq params,
in CORBA::ExceptionDefSeq exceptions,
in CORBA::ContextIdsSeq contexts,
in QoSAnnotation qos_annotation);
}

Flows

enum FlowDirection{FLOW_IN, FLOW_OUT};

interface FlowDef: CORBA::Contained{
  attribute FlowDirection direction;
  attribute MediaSpecificationDef media_type_def;
  readonly attribute TypeCode media_type;
  readonly attribute boolean qos_constrained;
  attribute QoSAnnotationDef qos_annotation;
};

struct FlowDescription{
  CORBA::Identifier name;
  CORBA::RepositoryId id;
  CORBA::RepositoryId defined_in;
  CORBA::VersionSpec version;
  FlowDirection direction;
  TypeCode media_type;
  boolean qos_constrained;
  QoSAnnotationDescription qos_annotation;
};

Stream interfaces

interface StrInterfaceDef;
typedef sequence <StrInterfaceDef> StrInterfaceDefSeq;
typedef sequence <FlowDescription> FlowDescriptionSeq;

interface StrInterfaceDef: InterfaceDef{
  attribute StrInterfaceDefSeq base_interfaces;

  struct FullStrInterfaceDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    FlowDescriptionSeq flows;
    CORBA::AttrDescriptionSeq attributes;
    CORBA::RepositoryIdSeq base_interfaces;
    boolean is_abstract;
    TypeCode type;
  };
  FullStrInterfaceDescription describe_interface();

  FlowDef create_flow(
in CORBA::RepositoryId id,
in CORBA::Identifier name,
in CORBA::VersionSpec version,
in FlowDirection direction,
in MediaSpecificationDef media_spec,
in QoSAnnotation qos_annotation);
};
```
Signals

```csharp
enum SignalDirection {SIG_IN, SIG_OUT};

struct ValueDescription{
    CORBA::Identifier name;
    TypeCode type;
};
typedef sequence <ValueDescription> ValueDescriptionSeq;

interface SignalDef: CORBA::Contained{
    attribute SignalDirection direction;
    attribute ValueDescriptionSeq values;
    readonly attribute boolean qos_constrained;
    attribute QoSAnnotationDef qos_annotation;
};

struct SignalDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    SignalDirection direction;
    ValueDescriptionSeq values;
    boolean qos_constrained;
    QoSAnnotationDescription qos_annotation;
};

Signal interfaces

interface SigInterfaceDef;
typedef sequence <SigInterfaceDef> SigInterfaceDefSeq;
typedef sequence <SignalDescription> SignalDescriptionSeq;

interface SigInterfaceDef: InterfaceDef{
    attribute SigInterfaceDefSeq base_interfaces;
    struct FullSigInterfaceDescription{
        CORBA::Identifier name;
        CORBA::RepositoryId id;
        CORBA::RepositoryId defined_in;
        CORBA::VersionSpec version;
        SignalDescriptionSeq signals;
        CORBA::AttrDescriptionSeq attributes;
        CORBA::RepositoryIdSeq base_interfaces;
        boolean is_abstract;
        TypeCode type;
    };
    FullSigInterfaceDescription describe_interface();
    SignalDef create_signal(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in SignalDirection direction,
        in ValueDescriptionSeq values,
        in QoSAnnotation qos_annotation);
};

Components

interface ComponentDef: CORBA::Contained{
};

/* Primitive components */

struct PrimInterface{
    InterfaceDef interf_type;
    CORBA::Identifier interf_name;
};
typedef sequence<PrimInterface> PrimInterfaceSeq;
typedef sequence<octet> ImplemCode;
```
interface PrimComponentDef: ComponentDef{
    attribute string implemen_name;
    attribute PrimInterfaceSeq interfaces;
    readonly attribute ImplemCode implem;

    PrimInterface get_interface_by_name(in CORBA::Identifier interf_name);
};

struct PrimInterfaceDescription{
    CORBA::Identifier interf_name;
    CORBA::RepositoryId interf_type_id;
};
typedef sequence<PrimInterfaceDescription> PrimInterfaceDescriptionSeq;

struct PrimComponentDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    string implemen_name;
    PrimInterfaceDescriptionSeq interfaces;
};

/* Composite components */

struct InternalComponent{
    BaseComponentDef component_type;
    CORBA::Identifier component_name;
};
typedef sequence<InternalComponent> InternalComponentSeq;

struct GraphNode{
    CORBA::Identifier component_name;
    CORBA::Identifier interface_name;
};

struct LocalBindingDcl{
    GraphNode node_a;
    GraphNode node_b;
};
typedef sequence<LocalBindingDcl> ObjectGraph;

struct CompInterface{
    InterfaceDef interf_type;
    CORBA::Identifier interf_name;
    GraphNode interf_expose;
};
typedef sequence<CompInterface> CompInterfaceSeq;

interface CompComponentDef: ComponentDef{
    attribute InternalComponentSeq internal_components;
    attribute ObjectGraph object_graph;
    attribute CompInterfaceSeq interfaces;

    CompInterface get_interface_by_name(in CORBA::Identifier interf_name);
};

struct CompInterfaceDescription{
    CORBA::Identifier interf_name;
    CORBA::RepositoryId interf_type_id;
    GraphNode interf_expose;
};
typedef sequence<CompInterfaceDescription> CompInterfaceDescriptionSeq;

struct InternalComponentDescription{
    CORBA::Identifier component_name;
    CORBA::RepositoryId component_type_id;
};
typedef sequence<InternalComponentDescription> InternalComponentDescriptionSeq;

struct ComponentDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    InternalComponentDescriptionSeq internal_components;
Appendix C – IDL / ODL Definitions of the Prototype

```c
ObjectGraph object_graph;
CompInterfaceDescriptionSeq interfaces;

Binding roles

enum MatchingRule {STRICT, SUBTYPE, PARTIAL};
typedef sequence<CORBA::Identifier> EssentialInteractions;

struct Cardinality{
    unsigned long min;
    unsigned long max;
};
typedef sequence<CORBA::Identifier> RoleNameSeq;

struct Dependencies{
    RoleNameSeq required;
    RoleNameSeq excluded;
};

interface BaseRoleDef: CORBA::Contained{
    attribute MatchingRule matching_rule;
    attribute Cardinality cardinality;
    attribute Dependencies causal_depend;
    attribute EssentialInteractions essential_interactions;

    boolean satisfied_by(in InterfaceDef interf_def);
};

/* Roles for primitive bindings */

struct PrimTargetInterface{
    InterfaceDef interf_type;
};

interface PrimRoleDef: BaseRoleDef{
    attribute PrimTargetInterface target_interface;
};

struct PrimRoleDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    CORBA::RepositoryId target_interface;
    MatchingRule matching_rule;
    EssentialInteractions essential_interactions;
    Cardinality cardinality;
    Dependencies causal_depend;
};

/* Roles for composite bindings */

struct TargetInterface{
    InterfaceDef interf_type;
    GraphNode interf_expose;
};

interface RoleDef: BaseRoleDef{
    attribute TargetInterface target_interface;
    attribute InternalComponentSeq components;
    attribute ObjectGraph configuration;
};

struct TargetInterfaceDescription{
    CORBA::RepositoryId interf_type_id;
    GraphNode interf_expose;
};

struct InternalComponentDescription{
    CORBA::Identifier component_name;
    CORBA::RepositoryId component_type_id;
};

typedef sequence<InternalComponentDescription> InternalComponentDescriptionSeq;

struct RoleDescription{
```

257
CORBA::Identifier name;
CORBA::RepositoryId id;
CORBA::RepositoryId defined_in;
CORBA::VersionSpec version;
InternalComponentDescriptionSeq components;
TargetInterfaceDescription target_interface;
MatchingRule matching_rule;
EssentialInteractions essential_interactions;
Cardinality cardinality;
Dependencies causal_depend;
ObjectGraph configuration;
}

Bindings

interface BaseBindingDef: CORBA::Contained, CORBA::Container{

/* Primitive bindings */

interface PrimBindingDef: BaseBindingDef{
    attribute string implem_name;
    readonly attribute ImplemCode implem;

    PrimRoleDef create_role(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in PrimTargetInterface target_interf,
        in MatchingRule matching_rule,
        in Cardinality cardinality,
        in Dependencies causal_depend,
        in EssentialInteractions essential_interactions);
}

typedef sequence<PrimRoleDescription> PrimRoleDescriptionSeq;

struct PrimBindingDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    string implementation;
    PrimRoleDescriptionSeq roles;
};

/* Composite (open) bindings */

struct ControlInterface{
    InterfaceDef interf_type;
    CORBA::Identifier interf_name;
    ComponentDef implem_component;
    CORBA::Identifier exposed_interface;
};

typedef sequence<ControlInterface> ControlInterfaceSeq;

struct InternalBinding{
    BindingDefBase binding_type;
    CORBA::Identifier binding_name;
};

typedef sequence<InternalBinding> InternalBindingSeq;

interface BindingDef: BaseBindingDef{
    attribute ControlInterfaceSeq control_interfaces;
    attribute InternalBindingSeq internal_bindings;

    RoleDef create_role(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in InternalComponentSeq components,
        in TargetInterface target_interf,
        in ObjectGraph configuration,
        in MatchingRule matching_rule,
        in Cardinality cardinality,
        in Dependencies causal_depend,
Appendix C – IDL / ODL Definitions of the Prototype

```c++
#include <EssentialInteractions>

struct ControlInterfaceDescription{
    CORBA::RepositoryId interf_type_id;
    CORBA::Identifier interf_name;
    CORBA::RepositoryId implem_component;
    CORBA::Identifier exposed_interf;
};
typedef sequenceControlItemDescription> ControlInterfaceDescriptionSeq;

struct InternalBindingDescription{
    CORBA::RepositoryId binding_type;
    CORBA::Identifier binding_name;
};
typedef sequenceInternalBindingDescription> InternalBindingDescriptionSeq;
typedef sequenceRoleDescription> RoleDescriptionSeq;

struct BindingDescription{
    CORBA::Identifier name;
    CORBA::RepositoryId id;
    CORBA::RepositoryId defined_in;
    CORBA::VersionSpec version;
    ControlInterfaceDescriptionSeq control_interfaces;
    InternalBindingDescriptionSeq internal_bindings;
    RoleDescriptionSeq roles;
};

Redefine feature: Container

interface Container: CORBA::Container{
    OpInterfaceDef create_op_interface(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in OpInterfaceDefSeq base_interfaces,
        in boolean is_abstract);

    StrInterfaceDef create_str_interface(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in StrInterfaceDefSeq base_interfaces,
        in boolean is_abstract);

    SigInterfaceDef create_sig_interface(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in SigInterfaceDefSeq base_interfaces,
        in boolean is_abstract);

    Components::PrimComponentDef create_prim_component(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in string implem_name,
        in Components::PrimInterfaceSeq interfaces);

    Components::CompComponentDef create_component(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in Components::InternalComponentSeq internal_components,
        in Components::LocalBindingDclSeq obj_graph,
        in Components::CompInterfaceSeq interfaces);

    Bindings::PrimBindingDef create_prim_binding(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in string implem_name);

    Bindings::CompBindingDef create_binding(
        in CORBA::RepositoryId id,
        in CORBA::Identifier name,
        in CORBA::VersionSpec version,
        in string implem_name);

    ControlInterfaceDescriptionSeq essential_interactions);

    InternalBindingDescriptionSeq internal_bindings;
    RoleDescriptionSeq roles;
};
```

259
in CORBA::Identifier name,
in CORBA::VersionSpec version,
in Bindings::ControlInterfaceSeq control_interfaces,
in Bindings::InternalBindingSeq internal_bindings);

Redefined feature: Repository

interface Repository: CORBA::Repository{
    MediaTypeSystemDef create_media_type_system{
        in CORBA::RepositoryId id,
in CORBA::Identifier name,
in CORBA::VersionSpec version);
    QoSAttrBaseDef create_qos_attr_base{
        in CORBA::RepositoryId id,
in CORBA::Identifier name,
in CORBA::VersionSpec version,
in QoSAttrDescriptionSeq attr_descrs
    };
    QoSAnnotationDef create_qos_annotation{
        in QoSAttrBase qos_base,
in QoSAttributeSeq attrs
    };
    CORBA::RepositoryId new_version(in any type_description);
};

/* End of the MetaORB module */

C.4 Meta-objects: interfaces and component types

Overview and basic definitions

In this section, the complete description of the interfaces of meta-objects, which correspond to the default meta-object protocols, are defined. In addition, auxiliary definitions are also provided. The interface definitions are presented in ODL syntax.

#include <CORBA.idl>
#include <MetaORB.idl>

module Reflection{

typedef sequence<string> NameSeq;

// Enumeration of the status codes for the exceptions raised by operations in this
// module.
enum StatusCodes{OPERATION_NOT_APPLICABLE, BAD_ARGUMENTS, ARCH_UNDEFINED_ERROR,
ARCH_INCOMPATIBLE_INTERFS, ARCH_INEXISTENT_COMP, ARCH_INVALID_TYPE,
ARCH_COULD_NOT_CREATE, ARCH_INCOMPATIBLE_ADJ_INTERFS,
ARCH_INVALID_POSITION, ARCH_INEXISTENT_BINDING,
ARCH_INEXISTENT_ROLE, ARCH_INEXISTENT_INTERF
};

// Exception raised by all operations in this module in case of errors of
// any nature. The code member should be left optional.
except Exception ReflectionException{
    string text;
    StatusCodes code;
};
Interface Discovery meta-objects

```c
struct InterfMapping{
    string current_interf_uname;
    string new_interf_name;
};
typedef sequence<InterfMapping> InterfMappingSeq;

/* INTERFACE OF INTERFACE DISCOVERY (ID) META-OBJECTS: CORRESPONDS TO THE INTERFACE * DISCOVERY MOP */
interface InterfDiscMeta_interf {
    //---- Interface introspection:
    // get a list with the names of all interfaces of the base-level object
    // (which can be either a component or a binding)
    NameSeq get_interf_names();

    //---- Introspection on binding endpoints:
    // get a list with the names of all endpoints (i.e. role instances) of the
    // base-level binding
    MetaORB::EndpointList get_endpoints()
        raises (ReflectionException); // in case the base-level object is not a binding

    // get repository id of the type of the target interface of a binding role
    CORBA::RepositoryId get_target_interf_type_id(
        in string role_name)
        raises (ReflectionException); // in case the base-level object is not a binding

    //---- Type manipulation:
    // get the id of type of the base-level component or binding
    CORBA::RepositoryId get_type_id();

    // check substitutability of the base-level component (by a component of the type
    // identified by the given repository identifier)
    InterfMappingSeq replaceable_by(
        in CORBA::RepositoryId comp_def_id)
        raises (ReflectionException);}
};

/* COMPONENT TYPE FOR INTERFACE DISCOVERY META-OBJECTS */
primitive component InterfDiscMeta_comp {
    implementation: InterfDiscMeta_comp_impl;
    interfaces:
        InterfDiscMeta_interf interfdisc_meta_interf;
};
```

Interface meta-objects

```c
union FullInterfaceDescription
    switch (MetaORB::InterfaceStyle)
    {
    case MetaORB::STYLE_OPERATIONAL:
        MetaORB::OpInterfaceDef::FullOpInterfaceDescription op_interf_descr;
        break;
    case MetaORB::STYLE_STREAM:
        MetaORB::StrInterfaceDef::FullStrInterfaceDescription str_interf_descr;
        break;
    case MetaORB::STYLE_SIGNAL:
        MetaORB::SigInterfaceDef::FullSigInterfaceDescription sig_interf_descr;
    }

union IntDescription
    switch (MetaORB::InterfaceStyle)
    {
    case MetaORB::STYLE_OPERATIONAL:
        MetaORB::OperationDescription op_descr;
        break;
    case MetaORB::STYLE_STREAM:
        MetaORB::FlowDescription flow_descr;
        break;
    case MetaORB::STYLE_SIGNAL:
        MetaORB::SignalDescription sig descr;
    }

typedef sequence <any> ValueList;

/* INTERFACE OF INTERFACE META-OBJECTS: CORRESPONDS TO THE INTERFACE MOP */
interface InterfMeta_interf{
    //---- Generic Introspection:
    // get unique interface name
    string get_interf_name();
};
```
// get interface style (operational, stream or signal)
string get_interf_style();

// get full interface description
FullInterfaceDescription get_interf_descr();

//----- Attribute Introspection:
// get list with the names of all attributes
NameSeq get_attr_list();

// get type of an attribute (as a type code)
MetaORB::TypeCode get_attr_type(in string attr_name);

//----- Interaction Introspection:
// get list with the names of all interactions
NameSeq get_interaction_list();

// get interaction names based on the causality (the result should be interpreted
// according to the style of the base-level interface, i.e., as a list of
// operation, flow or signal names):
NameSeq get_prov_interactions();
NameSeq get_req_interactions();

// get interaction description
MetaORB::IntDescription get_interaction_descr(in string interaction_name);

//----- Direct (dynamic) Access:
// get and set the value of an attribute:
any get_attr_value(in string attr_name)
raises(ReflectionException);
void set_attr_value(in string attr_name, in any value)
raises(ReflectionException);

// invoke an operation (only for operational interfaces):
// args = "in" and "inout" arguments;
// return_value = tuple with result + "out" and "inout" arguments;
// raises exception if the invocation is invalid or if it raises an exception
ValueList invoke(in string op_name, in ValueList args)
raises(ReflectionException);

//----- Type access:
// get the type of the base interface
CORBA::RepositoryId get_type_id();

};//

/* COMPONENT TYPE FOR INTERFACE META-OBJECTS */
primitive component InterfaceMeta_comp{
    implementation: EncMeta_comp_impl;
    interfaces:
        InterfaceMeta_interf interf_meta_interf;
};

Architecture meta-objects

struct BoundComp{
    string interf_name;
    MetaORB::GraphNode other_node;
};
typedef sequence <BoundComp> BoundCompSeq;

struct InsertLocation{
    string interf_uname_a;
    string interf_uname_b;
};

/* INTERFACE OF ARCHITECTURE META-OBJECTS: CORRESPONDS TO THE ARCHITECTURE MOP */
interface ArchMeta_interf {

    //----- Generic introspection on the object graph:
    // get the complete object graph
    MetaORB::ObjectGraph get_obj_graph();

    // get a list with the names of all internal components
    NameSeq get_internal_comps();

}
// get a list with information about the components that are bound to the named
// component, including component uname and the unames of the bound interfaces
BoundCompSeq get_bound_comps(in string comp_uname);

//----- Introspection for binding objects:
// get the abstract configuration of a binding role (as defined in the type)
MetaORB::ObjectGraph get_role_config(in string role_name)
  raises (ReflectionException);

// get the concrete configuration of a binding endpoint
MetaORB::ObjectGraph get_endp_config(in string role_name,
  in string capsule_name)
  raises (ReflectionException);

// get a list with the unames of all nested bindings
NameSeq get_internal_bindings()
  raises (ReflectionException);

//----- Low-level intercession: manipulation of local bindings (based on the
// concrete object graph)
// establish a local binding between two interfaces
void local_bind(in string interf_uname_1, in string interf_uname_2)
  raises (ReflectionException);

// break an existing local binding between two interfaces
void break_local_binding(in string interf_uname_1, in string interf_uname_2)
  raises (ReflectionException);

//----- Intercession based on the concrete object graph (for components and
// bindings):
// insert a new component at a specific position in the configuration,
// between two interfaces that are currently bound to each other (as local
// bindings are always two-way, it is not possible to specify a position
// other than between two interfaces)
void insert_component(in CORBA::RepositoryId new_comp_type_id,
  in string new_comp_name,
  in InsertLocation location)
  raises (ReflectionException);

// NOTE: There are many other meaningful positions for insertion of new
// components that could be explored (e.g., insertion between any number of
// dangling interfaces). Currently, however, the MOP is limited to insertion
// between two interfaces.

// remove a component from a specific location in the configuration
void remove_component(in string comp_uname)
  raises (ReflectionException);

// replace a component with a new one
void replace_component(in string old_comp_uname,
  in string new_comp_name,
  in CORBA::RepositoryId new_comp_type_id)
  raises (ReflectionException);

//----- Intercession based on the concrete object graph (only for binding
// objects):
// replace an existing internal binding with another one of the given
// type (retaining the binding's name)
void replace_binding(in string old_binding_name,
  in CORBA::RepositoryId new_binding_type_id)
  raises (ReflectionException);

// add a new, non-connected component to a particular endpoint of a binding
void add_binding_component(in CORBA::RepositoryId new_comp_type_id,
  in string new_comp_name,
  in ORBcore::EndpointInfo endpoint)
  raises (ReflectionException);

//----- Intercession for components:
// add a new, non-connected component to the configuration of the base-level
// component
void add_component(in CORBA::RepositoryId new_comp_type_id,
  in string new_comp_name)
  raises (ReflectionException);

//----- Intercession – role-based, with effect on all instances of the role
// and semantics equivalent to their endpoint-based counterparts
// (only for binding objects):
// add a new, non-connected component to the configuration of a role
void role_add_component(in string role_name,
in CORBA::RepositoryId new_comp_type_id,
in string new_comp_name)
    raises (ReflectionException);

// establish a local binding between two interfaces (represented by node_a
// and node_b) of a role configuration
void role_local_bind(in string role_name,
in MetaORB::GraphNode node_a,
in MetaORB::GraphNode node_b)
    raises (ReflectionException);

// break an existing local binding between two interfaces (represented by node_a
// and node_b) of a role configuration
void role_break_local_binding(in string role_name,
in MetaORB::LocalBindingDcl lbind)
    raises (ReflectionException);

// insert a new component in the configuration of a role, between the interfaces
// represented by node_a and node_b
void role_insert_component(in string role_name,
in CORBA::RepositoryId new_comp_type_id,
in string new_comp_name,
in MetaORB::LocalBindingDcl lbind)
    raises (ReflectionException);

// remove a component from a role configuration
void role_remove_component(in string role_name,
in string comp_name)
    raises (ReflectionException);

// replace a component in a role configuration (retaining the component name)
void role_replace_component(in string role_name,
in string old_comp_name,
in CORBA::RepositoryId new_comp_type_id)
    raises (ReflectionException);

//---- Encapsulation access:
// get the name of the internal component exposing an interface
string get_interf_exposer(in string interf_name)
    raises (ReflectionException);

// expose an interface of an internal component as an external interface of
// the base-level component (not supported for bindings, currently)
void expose_interf(in string ext_interf_name,
in string interf_exposer,
in string exposed_interf)
    raises (ReflectionException);

//---- Type access: introspection
// get the type id of the base-level component or binding object (note that the
// returned type may be out-dated if there was any intercession)
CORBA::RepositoryId get_type_id();

//---- Type access: intercession
// create a new type definition based on the current configuration of the base-
// level object; change the type of the base-level object to this new type and
// return its new repository identifier
CORBA::RepositoryId commit_type()
    raises (ReflectionException);
References


Annapolis, MD USA, May 1998.


Gamma, E., R. Helm, R. E. Johnson and J. Vlissides (1995). Design Patterns, Elements of Object-Oriented Software, Addison-Wesley.


OMG (2001a). The Common Object Request Broker: Architecture and Specification - Revision 2.4.2 (CORBA v2.4.2). Object Management Group, Needham, MA USA.

OMG (2001b). Interceptors Published Draft with CORBA 2.4+ Core Chapters (Portable Interceptors specification) (OMG Document ptc/2001-03-04). Object Management Group, Needham, MA USA.


