OOPP

A Reflective Middleware Platform
including
Quality of Service Management

Anders Andersen

February 2, 2002

A dissertation submitted in partial fulfillment
for the degree of Doctor Scientiarum

Department of Computer Science
Faculty of Science
University of Tromsø

To Brynjar and Merete

In the memory of my mother Tordis Helene Andersen

This dissertation was typeset with the LaTeX macro package for TeX. The figures was drawn using MetaPost, the bibliography was prepared using BibTeX, and the author/editor index was prepared using Authidx. The fonts used are Concrete Roman, Computer Modern Sans and Euler.

A dissertation produced with the TeX typesetting system

Abstract

Middleware has emerged in distributed systems to present a unified programming model to application programmers and mask out the problems of heterogeneity and distribution. The reason to introduce middleware is to ease the process of creating complex (distributed) applications. Software developers recently have observed a related change in the software industry. This change is called software components (or just components) and it has changed the way software can be developed, deployed and maintained. A component is self-contained code that can be independently developed and deployed. This ease the process of building a complex application containing a composition of different components (possible from different providers) that interacts to provide the functionality of the application.

Present middleware are made available to the application programmer through a (static) Application Programming Interface (API). Different application domains have different demands made upon the middleware platform. The results are complex and huge middleware implementations providing a lot of functionality with few possibilities to change and adapt its behaviour. Continuous media streams need communication protocols quite different from traditional remote procedure calls. Some applications might have to adapt to different net characteristics at run-time. Such special needs are rarely supported in current middleware implementations. The design of the next generation middleware should adopt an open engineering approach for the support of configuration and adaption. Reflection provides an principled (as opposed to ad hoc) means of achieving such open engineering.

The OOPP platform presented in this work is an implementation of an reflective middleware platform with an expressive programming model. It is an investigation into how to implement such a platform, and an investigation into what programming model it should provide.

Quality of service management is essential for many application domains. An open engineering approach provides the mechanisms to manipulate (adapt) the middleware for given quality of service demands. A quality of service management system can use these mechanisms to perform its management tasks. However, it has to implement the policies for what to do when based on observation of the behaviour of the system made available with an open engineering approach.

Describing management policies in a formal language has a lot of advantages. The obvious one is the possibility to formally reason about the consequences of a given policy applied on a (formally described) system. A formal description can also be interpreted and executed on a computer. The possibility to actually interpret (and run) a formally described management policy can in fact be used in a running system. In OOPP, quality of service management is successfully introduced in the running system based on components with their behaviour (management policies) described in the formal language of timed automata. The actual timed automata descriptions are either generated by tools (for example from temporal logic describing the management constraints or from simulation of the running system) or made manually by programmers specialised in such tasks.

The OOPP platform has been successfully used in several research activities looking into special enhancements to the platform. It has also been used successfully to build applications that needs a system that can adapts its behaviour.

April 5, 2002: Page iii and vii updated. Empty pages (page 133 and 134) removed.

Acknowledgements

I have to thank a lot of people that supported and provided feedback to my work on this project. However, every error and weakness introduced in this work are my own responsibility.

Professor Frank Eliassen has been the supervisor of this project. He has played that role perfectly. He has guided the work and helped high-lightning the important research issues and results. His contributions were of great importance, especially in the final stage of the project.

Professor Gordon Blair brought me in to the Open-ORB project at its early stage (before it was named Open-ORB), and introduced me to the concept of reflection. He has provided a lot of ideas and has been a great discussion partner (even when he runs). The OOPP project would never have existed if it was not for his contributions.

The people at NORUT IT, the University of Tromsø, and Lancaster University have provided the right environment to do this work. At NORUT IT I will especially mention Henning Sundby, Guðmundur Jökulson, Asgeir Finnseth, Dag Diesen and Arne-Wilhelm Theodorsen. At the University of Tromsø I want especially to highlight the co-operation with Tage Stabell-Kulø (providing all the difficult questions starting with "why"), Hans Ole Rafaelsen and the people of the ODS group. At Lancaster University I have to mention Michael Papathomas, Geoff Coulson, Lynne Blair, Fábio Moreira Costa, Hector Duran Limon, Katia B. Saikoski and David Sánchez Gancedo (the important first user of OOPP). You all made the Open-ORB project a great home for OOPP. Many more from NORUT IT, the University of Tromsø, Lancaster University, the University of Oslo and UniK at Kjeller should also be mentioned. You know who you are. To meet and work with you has been great.

Friends and family outside the professional life are an important resource. Thank you for being there. And to my father: Thank you for the support from you and my mother (Til min far: Takk for at du og mor alltid støttet meg).

Finally, and most important, Brynjar and Merete. You make me happy and you make me feel good. You are great. This work is dedicated to you.

April 5, 2002, Anders Andersen

This project was funded by the Norwegian Research Council and NORUT IT through the ELDORADO project (Project 101048/420). The project has also been supported by a NATO Science Fellowship (through the Norwegian Research Council, Project 116968/410), the EPSRC Grant GR/L31609 entitled "A Specification Architecture for the Validation of Real-time and Stochastic Quality of Service" (trough the Lancaster University), and the University of Tromsø.

Contents

List of figures

List of tables

List of listings

Abbreviations

API Application Programming Interface

COM Component Object Model

CORBA Common Object Request Broker Architecture

DCE Distributed Computing Environment

DCOM Distributed Component Object Model

EJB Enterprise JavaBeans

GIOP General Inter-ORB Protocol

IDL Interface Definition Language

IIOP Internet Inter-ORB Protocol

ISO International Organisation for Standardisation

MOP Meta Object Protocol

MPEG Moving Picture Experts Group

ODP Open Distributed Processing

OMG Object Management Group

OOPP Open-ORB Python Prototype

ORB Object Request Broker

OSF	Open Software Foundation
PDA	Personal Digital Assistant
QoS	Quality of Service
RMI	Remote Method Invocation
RM-ODP	The Reference Model for Open Distributed Processing
RPC	Remote Procedure Call
TINA	Telecommunication Information Networking Architecture
URL	Uniform Resource Locators

Part I

Context

Chapter 1

Introduction

Middleware has emerged in distributed systems to present a unified programming model to application programmers and mask out the problems of heterogeneity and distribution. The 'middle' part of middleware has two possible explanations. One is that the middleware is a layer between the application and the underlying operating system and network protocols. The other is related to client-server oriented applications where the middleware is located between the clients and their servers. However, the reason to introduce middleware is to ease the process of creating complex (distributed) applications.

Software developers recently have observed a related change in the software industry [80]. This change is called software components (or just components) and it has changed the way software can be developed, deployed and maintained. A component is self-contained code that can be independently developed and deployed. This ease the process of building a complex application containing a composition of different components (possible from different providers) that interacts to provide the functionality of the application.

Current middleware and component technology have none or little support for multimedia and mobility. In some cases, a set of extensions providing a limited support are provided. However, none treats multimedia and mobility as a fundamental concept of their infrastructure.

1.1 Trends in middleware

The term middleware appeared in the literature following the trends of client-server architectures. Current middleware has been influenced by the early work on RPC [18] and the DCE framework from OSF [65]. More recently, RM-ODP [60], CORBA [88], Java RMI [83], DCOM/.NET [84, 104] and other [105, 36] have influenced the design and implementation of middleware platforms. The following gives a short introduction to some of the recent trends that have influenced this project.

1.1.1 Interfaces and components

A common trend in distributed system technology is to adopt an approach based on interfaces. Interfaces describes a set of services provided (or requested), usually in a language and platform independent way. CORBA and RM-ODP are examples of distributed platforms that have adopted this approach.

Software components are a similar approach to create or wrap self-contained code that can be developed and deployed independently. All interactions between components happen through a set of well defined interfaces. Microsoft's COM technology and SUN's JavaBeans are examples of such component models. A component is by Szyperski [103] viewed "as a unit of composition with contractually specified interfaces and explicit context dependencies only". Components are now an established technique used in the construction of complex applications. According to Szyperski, "a component can be deployed independently and is subject to third-party composition". This matches the needs for the construction of distributed applications perfectly. CORBA components, Enterprise JavaBeans (EJB) and DCOM are examples of such distributed component models.

1.1.2 Open engineering and reflection

Present middleware are made available to the application programmer through a (static) API (Application Programming Interface). Different application domains have different demands made upon the middleware platform. The results are complex and huge middleware implementations providing a lot of functionality with few possibilities to change and adapt its behaviour. Continuous media streams need communication protocols quite different from traditional remote procedure calls. Some applications might have to adapt to different net characteristics at run-time. Such special needs are rarely supported in current middleware implementations.

The emerge of applications that need support for continuous media, mobility, quality of service management and so on, has pushed the middleware community to look in to more adaptive middleware solutions. The traditional black-box philosophy where the application programmer has no possibility to change (or specify) the underlying behaviour of the middleware is too limited for many application domains. *One-size-fits-all* is not good enough. Some applications either need a specific solution for its specific characteristics or a configurable and adaptive solution that fits a wider range of applications.

Configuration and adaptive features have been introduced in some of the current middleware implementations. The problem with these solutions are that they only provide limited possibilities to configure and adapt the behaviour of the middleware platform. The design of the next generation middleware should adopt an open engineering approach for the support of configuration and adaption. Reflection provides an principled (as opposed to ad hoc) means of achieving such open engineering.

Reflection initially emerged in the programming language community as an important technique to introduce more flexibility and openness into the design of programming languages. Later, reflection has been adopted in several other types of systems. Graphical user interfaces and operating systems are examples of systems where reflection has been adopted with success. Recently, reflection has been introduced in several middleware projects.

1.1.3 Quality of service management

Quality of service management is essential for many application domains. An example is a continuous media stream where the quality of the connection between the stream source and stream sink needs to be managed. This may involve functions like buffer management policies, media quality, and resource allocations.

An open engineering approach provides the mechanisms to manipulate (adapt) the middleware for given quality of service demands. A quality of service management system can use these mechanisms to perform its management tasks, but it has to implement the policies for what to do when based on observation of the behaviour of the system made available with the open engineering approach.

Management policies are often described in a formal language [24]. This has a lot of advantages. The obvious one is the possibility to formally reason about the consequences of a given policy applied on a (formally described) system. A formal description can also be interpreted and executed on a computer simulating the behaviour of the system with the given management policy. The possibility to actually interpret (and run) a formally described management policy could also be used in an actual running system.

1.2 Research issues

The first goal of this project is to investigate how to implement an expressive reflective middleware platform and what reflective features it should provide. An important issue is the expressiveness and ease of use of the provided programming model. The second goal is to investigate how quality of service management functions based on formal descriptions could be applied to this platform. The list of research issues in this project follows below.

1.2.1 Implement a reflective middleware platform

Section 1.1.2 contains some motivation for a reflective middleware platform. Is it possible to implement a middleware platform that fulfils the demands behind the given motivation? How is such a middleware platform implemented? What are the reflective features needed to fulfil the demands? Will the implementation be efficient enough for a wide range of applications? And finally, is the resulting programming environment expressive and easy-to-use from the perspective of the programmer?

1.2.2 Introduce QoS management functions

Section 1.1.3 contains some motivation for quality of service management functions based on formal descriptions and open engineering. Is it possible to implement expressive management functions in the experimental reflective middleware platform based on formal descriptions of the management policies and to use the reflective features to perform the management tasks? And if so, is this approach flexible enough for a wide range of application domains?

1.3 Methodology

The work described in this document is the design and implementation of a middleware platform based on the new ideas and trends described above. This work is divided in two main parts: (i) the design and implementation of an experimental reflective middleware platform, and (ii) the introduction of dynamic quality of service management in this platform. The purpose of this project is to investigate the research issues from Section 1.2. The selected approach can be summarised in the following sentence:

> *Design and implement an experimental component-based reflective middleware platform and introduce dynamic quality of service management in this environment based on formal specification techniques.*

The designed and implemented middleware platform presented is given the name the *Open-ORB Python Prototype* (OOPP). This work is a contribution to the Open-ORB project at Lancaster University.

1.3.1 Elaboration

The middleware platform should provide an environment for distributed components. This includes managed address spaces for the components to exist in, and mechanisms to easily interact with local and remote components. It should also provide other necessary features of middleware platforms. This includes naming, a way to locate and get access to (remote) components, and mechanisms to create and deploy (and destroy) components.

Support for continuous media is also an important issue. This should be reflected in the provided programming model and in the underlying infrastructure. Continuous media needs quality of service management functions and support for media synchronisation. The quality of service provided will focus on dynamic quality of service management. Dynamic quality of service management is concerned with the run-time monitoring and control of services.

Quality of service management will be based on existing tools and specification techniques already in use in the Open-ORB and V-QoS projects at Lancaster University.

Reflection should be applied to all components. This includes application components and components that is a part of the implementation of the middleware platform itself.

1.3.2 Delimitation

The middleware platform presented in this work should not be regarded as a complete middleware platform ready to use in the software industry. Its purpose is to demonstrate some key features of a reflective middleware platform, to show how dynamic quality of service management can be introduced in such an environment, and to be a tool for further research on reflective middleware.

Static quality of service management is not given a high priority in this work. Static quality of service management is concerned with the establishment of the services. This includes quality of services specification, quality of services negotiation, admission control and resource reservation.

Performance and scaling will not be a major concern in this work. Performance is of course an important issue in a middleware platform where the demands from continuous media are used in the arguments for some of its key features. However, this work will focus on the characteristics of these features, how to introduce them in the middleware platform, and how to use them in dynamic quality of service management.

Security is completely ignored in the current version of OOPP. The consequences is that everybody with the appropriate knowledge can do whatever they want to do to every component of the applications and the middleware platform itself (even remotely).

Another important issue in middleware and component technology is standardisation. The idea is that if a middleware platform or a set of components are implemented in accordance to a given standard they could be used together with other products based on the same standard. For instance, two different implementations of the OMG CORBA platform should be able to interact through the standardised IIOP (Internet Inter-ORB Protocol) [89, 98]. The middleware platform presented here are not implemented according to any such standards.

1.3.3 A comment on the selected approach

This work has focused on how to implement (construct) a middleware platform with the characteristics and features described above. It has been done by early prototyping in a prototyping friendly language. Several prototypes have been implementing, each testing a limited set of features and ideas. The experience from these prototypes and related work are then, together with new ideas, used to implement the next generation prototype. Each new generation has been a step towards a more complete middleware platform. The final prototype described in this document is a collection of lessons learned through the process of prototyping, and development of new ideas.

This prototyping process can be described as the acquisition of knowledge needed to solve the given problem. Phillip G. Armour argues that software (the prototype) is a medium for the storage of knowledge [13] and that the product of the efforts to produce software is the knowledge contained in the software. It is easy to produce simple software and software similar to what we have produced before since it contains little new knowledge. The difficult part is to acquire knowledge used to solve new problems. The consequence is that software development is not a product-producing activity. It is a knowledge-acquiring activity. In [14] Phillip G. Armour adds:

> *"... prototyping acknowledges that our job is not to build a system, but to acquire knowledge."*

1.4 Results

The implementation of a reflective middleware platform has been demonstrated in a prototype of such a platform [10, 9]. The prototype has proven to fulfil the demands of applications needing support for adaptive behaviour from the platform itself. This document and other publications produced in this project discuss how to implement such a platform. Experiences with the reflective features provided by the prototype have shown the usefulness of these features and highlighted the most important such features. Experiences in related projects and activities using the prototype have shown that the resulting programming environment is usable and expressive from the perspective of the programmer.

Quality of service management functions based on automata have been successfully introduced and integrated in the platform [21]. This has shown that quality of service management based on formal descriptions can be implemented in such a platform. It has also shown that the combination of automata based management activities and the reflective features of the platform itself can provide a flexible and expressive management system. These dynamic management functions have proven to be usable and flexible enough for several cases with different dynamic non-functional behaviour.

1.4.1 Software

The reflective middleware platform prototype (OOPP) presented in the following text has been successfully implemented in Python [76]. It provides an expressive RM-ODP inspired programming model with different types of interfaces and bindings for signals, continuous media and standard method calls. The resulting programming model hides the complexity of distribution and underlying network protocols. It also provides some degrees of location and access transparency. A simple but easy-to-use naming service is also available.

Bindings between two remotely interfaces can be implicit or explicit. Implicit bindings are introduced without the knowledge of the programmer and hides the complexity of remote interaction. Explicit bindings makes it possible for the programmer to control the creation and configuration of a binding. The result is great flexibility where the programmer can control and configure the binding to the explicit needs of the given application.

Structural reflection is concerned with the contents of a component. Structural reflection is introduced in OOPP through the encapsulation and composition meta-models. These meta-models are successfully introduced in the platform. Under some circumstances will the introduction of meta-models result in side-effects like increased latency and decreased speed. The encapsulation meta-object of a given object can be used to inspect and manipulate the methods and attributes of this object. The encapsulation meta-object protocol provides a rich set of services for this kind of inspection and manipulation. The composition meta-object of a composite component can be used to inspect and manipulate the object-graph of this composite component. The composition meta-object protocol provides different types of services needed to inspect and manipulate the object-graph.

Finally, an automata based quality of services management mechanism has successfully been added to OOPP. These management components interact using signals. The underlying application and middleware platform are monitored and manipulated through their different meta-models. The behaviour of the management components can be described with a formal automata syntax. These management components behave according to their automata description. The automata description can be generated using different kinds of tools developed in related projects at Lancaster University.

OOPP is proven to work with somewhat complex examples. Typical examples includes synchronisation and management of audio streams, simple conference applications and continuous media streams adapting to the underlying network bandwidth.

1.4.2 Publications

Key publications

The OOPP platform itself is presented in «A Reflective Component-Based Middleware with Quality of Service Management» [10] (reprinted in Section C.1). OOPP is presented as reflective middleware platform with the possibility to inspect, adapt and extend the components of the system to satisfy the requirements of a given application. The paper also introduce quality of service management and how it is achieved using management components with different roles (monitors, strategy selectors and strategy activators). Timed automata is presented as one way of specifying the behaviour of such management components.

In «Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform» [21] (reprinted in Section C.2) the role of reflection in supporting the dynamic QoS management functions of monitoring and adaptation is considered. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies.

Other publications

Below follows a short description of work presented in other publications during this project. Work presented in recent publications are presented first.

In «Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures» [11] and «Arctic Beans: Flexible and Open Enterprise Component Architectures» [12] the Arctic Beans project is described. It is becoming increasingly apparent that existing middleware technologies are unable to accommodate the great diversity of application demands in modern distributed systems. This problem is particularly evident in emerging enterprise (or server-side) component architectures (such as Enterprise JavaBeans or the CORBA Component Model). Such architectures provide implicit support for distribution through the concept of a container. The main problem with this approach is that distribution management tends to be hidden from the developer. The Arctic Beans project is exposing distribution management, with the overall aim of developing a more open and flexible enterprise component architecture with intrinsic support for configuration, re-configuration and evolution.

«Flexible authentication and delegation for distributed components» [85] shows how modern security concepts can be configured and applied to a reflective middelware platform. Recent development in middleware has focused on a flexible and adaptable platform for application development and deployment, where reflection is one approach. Adding security will force constraints on any flexible and adaptable system, and will often be sacrificed if it seems like it will hamper use and deployment of the system. Instead of sacrificing the one for the other, we show how modern security concepts can be configured and applied to a reflective middelware platform, in such a way that the security platform is as flexible and dynamic as possible. This includes the ability to dynamically change both the security policy and policy specification at runtime, based on the interactions between middelware components.

In the «Formal Support for Dynamic QoS Management in the Development of Open Component-based Distributed Systems» paper [23] an aspect-oriented specification technique that supports the specification of component-based distributed systems is presented. Importantly, this technique also supports the synthesis of quality of service management components from particular aspects of the specification. We describe how, by using a tool to support our aspect-oriented environment, we can first specify and verify QoS management subsystems and then synthesise components that can be placed into a running system.

«The Design and Implementation of Open ORB 2» [27] discusses the design of the Open ORB 2 middleware platform, a next generation middleware platform that (i) is more configurable, (ii) is dynamically reconfigurable, and (iii) supports the longer term evolution of the design of the platform. These requirements are addressed through an approach based on reflection together with the use of component technology. Important issues addressed in the paper include the use of architectural constraints to maintain the integrity of reflective middleware platforms, together with studies of efficient implementation of such technology.

An overview of the overall OOPP architecture and programming model is presented in «OOPP: A Reflective Component-Based Middleware» [9]. OOPP is designed from the beginning with flexibility and adaptability in mind. This can be done by adopting an open engineering approach for the design of the middleware platform. OOPP implements a flexible and adaptable middleware based on the principle of reflection.

In «A Formal View of Aspects in the Development of Component-Based Distributed Systems» [22] an aspect-oriented specification technique that supports the specification of component-based distributed systems is presented. Importantly, this technique also supports the synthesis of quality of service management components from particular aspects of the specification. The paper contains a description of how to first specify and verify QoS management subsystems and then synthesise components that can be placed into a running system by using a tool to support our aspect-oriented environment. Focus is on dynamic QoS management functions, particularly on QoS monitoring and adaptation.

«A Principled Approach to Supporting Adaptation in Distributed Mobile Environments» [26] presents a principled approach to supporting adaptation through the use of reflection. More specifically, the paper introduces a language-independent reflective architecture featuring a per-object meta-space, the use of meta-models to structure meta-space, and a consistent use of object graphs to represent composite components.

The short paper «A reflective component-based middleware in Python» [8] presents the flexible and open Open-ORB Python Prototype (OOPP) with a focus on the challenges and demands from existing and new type of applications. This includes support for multimedia, real-time requirements, and increasingly mobility. The paper also has a focus on the consequences of selecting Python for the prototype implementation.

«Next Generation Middleware: Requirements, Architecture, and Prototypes» [46] proposes a next generation middleware architecture: an open engineering middleware platform that is run time configurable and allows inspection and adaptation of the underlying components. This architecture is based on the principle of reflection. The paper also reports on some existing research prototypes with a focus towards their suitability as next generation middleware.

The Application Programming Interface (API) of OOPP is presented in the technical report «The Open-ORB Python Prototype API» [7] from NORUT IT.

«The Role of Reflection in Supporting Dynamic QoS Management Functions» [20] builds on previous work on reflective middleware platforms. The paper considers the role of reflection in supporting the dynamic QoS management functions of monitoring and adaption. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies.

In «A Multi-Paradigm Specification Technique Supporting the Synthesis of QoS Management Components» [31], a multi-paradigm specification technique that supports the synthesis of quality of service management components is presented. A description on the usage of a tool that supports the multi-paradigm environment is included. The tool is first used to specify and verify QoS management subsystems and then used to synthesise components that can be placed into a running system.

«Separating Functional Behaviour and Performance Constraints: Aspect-Oriented Specification» [30] addresses the relationship between functional (qualitative) behaviour and the more quantitative nature of performance constraints. An approach based on aspect-oriented specification which exploits the diversity and power of existing formal specification languages is proposed. The approach is illustrated by specifying an example of an adaptive algorithm. The chosen example is characteristic of QoS management functions in the field of distributed multimedia systems.

«A Note on Reflection in Python 1.5» [6] is a report on reflection in Python 1.5. The report describes the open implementation in Python 1.5 and show how it can be used to achieve a reflective programming model. The report includes a description of an implementation of a reflection module for Python 1.5.

In the related ATOM project Python was used to prototype and experiment with the use of a novel concurrent object-oriented programming model called ATOM. In «Concurrent Object-Oriented Programming in Python with ATOM» [91] the model's main features was presented and their use for concurrent programming in Python were illustrated.

1.5 The rest of this document

The rest of this document is organised in four different parts: Part I (including this chapter) gives an overview of the context of this work. Part II presents the Open-ORB Python Prototype (OOPP) and how quality of service management is introduced in this platform. Part III contains evaluation and conclusion of the work presented earlier. Finally, part IV (the appendix) contains the listing of all authors/editors and publications referred to in this document, and a re-print of selected publications.

1.5.1 Part I, Context

This chapter gave an introduction to the work described in this thesis. In presented some major trends in middleware and listed the research issues of this project. A discussion of the methodology and the main results of this work have been presented. The chapter ends with this presentation of the contents of the thesis and the graphical and typographical conventions used in the thesis.

Chapter 2 gives an overview of the topics and related work covered by this project. First a presentation of middleware related topics and projects are presented. Then reflection and the usage of reflection are presented. Finally, topics related to quality of service management are presented.

Chapter 3 introduces the Open-ORB middleware platform from Lancaster University. The chapter starts with an introduction to reflective middleware and related work on reflective middleware. Then a presentation of the general principles and the programming model of Open-ORB is given. Finally the reflective features of Open-ORB is presented.

1.5.2 Part II, OOPP

Chapter 4 discusses design issues for the implementation of OOPP. First the relation between OOPP and RM-ODP is discussed. Then a discussion of the OOPP programming model and its infrastructure is presented. Finally a presentation of reflection and its meta-models in OOPP is given.

Chapter 5 describes the experimental implementation of OOPP. The structure of this chapter is similar to the structure of the previous chapter. It contains a description and discussion of the implementation of programming structures, the infrastructure and the meta-models of OOPP. The programming structures represents the core programming model of OOPP.

Chapter 6 shows how quality of service management is introduced in OOPP. First a presentation of the different management roles is given. Then the different managements objects or components are presented. Finally the actual management functions, including the management of management functions (meta management), are presented.

1.5.3 Part III, Discussion

Chapter 7 evaluates the work presented earlier. First some performance issues related to the provided flexibility is discussed. Then, the use of OOPP is illustrated through several examples. The chapter ends with some comments on the presented OOPP platform.

Chapter 8 concludes this work. First a summary is presented. Then a list of the major contributions and the future work is discussed. Finally the concluding remarks are given.

1.5.4 Part IV, Appendix

Appendix A is an index of all the authors and editors from publications referred to in this document. Appendix B is a listing of all the publications referred to in this document. Selected publications are re-printed in Appendix C.

1.6 Conventions

Example names Names in examples are printed in the euler font (like '$\mathfrak{a}$' and '$\mathfrak{b}$').

Code Source code are printed in the Computer Modern Sans font (like 'play').

Interfaces Interfaces are drawn with the symbol '$\perp$' (rotated in any directions).

Local bindings Local bindings are drawn with two connecting interfaces ('⊣⊢').

System components System components are usually drawn in dark grey ('●' or '●').

User components User components are usually drawn in light grey ('○' or '○').

Other objects Some low-level objects are drawn in white ('○').

Binding object Binding objects are 'long' objects with two (or more) interfaces ('⊢⬭⊣').

Implicit binding An implicit binding is (usually) drawn with the symbol '⊢—⊣'.

Operational binding Operational bindings can be marked with the symbol '⇄'.

Signal binding Signal bindings are marked with the symbol '⤳'.

Stream binding Stream bindings are marked with the symbol '∿'.

Attribute An attribute in an object is drawn with the symbol '⊙'.

Method A method in an object is drawn with the symbol '⊟'.

Loop A repeating (looping) activity is drawn with the symbol '↻'.

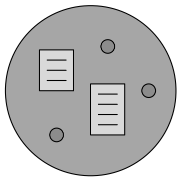
Timed loop A timed repeating activity is drawn with the symbol '↻'.

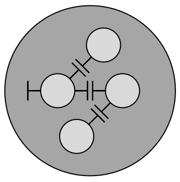
Queue A queue (typically a message queue) is drawn with the symbol '▥'.

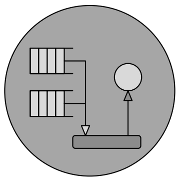
Manager A manager is drawn with the symbol '▬'.

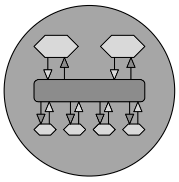
Abstract Resource An abstract resource is drawn with the symbol '⬭'.

Meta-models The four meta-models of an object are usually drawn like this:


Encapsulation


Composition


Environment


Resources

Chapter 2

Overview

This chapter gives an overview of and introduction to the different topics this project covers. It includes middleware, reflection, and quality of service management. Related work is also presented in this overview.

2.1 Middleware

Early programming models for distributed systems had a goal of hiding the lower level of the involved communication. Remote Procedure Call (RPC) [18] provides access to remote functionality (procedures) in a programming model similar to how local functionality (procedures) are accessed. The actual communication involving marshaling (packing the request and its data), data transfer (packed data on the wire) and unmarshaling (unpacking the request and its data) are hidden for the programmer. RPC only focuses on communication. Other system level functionality used by programmers are accessed directly using the services provided by the hosting system (operating system) of the program.

Middleware aims to provide a standard programming model covering all system functionality. The access to this functionality are provided on a higher level of abstractions. These abstractions should provide a programming model independent of the hosting system. Middleware supports the developers of distributed applications [68]. It provides a unified programming model and masks out problems related to distribution and heterogeneity. The services provided by the middleware is a standard programming environment and standard protocols [45, 16].

In middleware platforms like CORBA, DCOM/.NET, and Java RMI, the programs use the concept of *interfaces* (of objects) to describe and publish its services. Such interfaces are represented by the signature of its methods. A *signature* describes the arguments and return values of a given method. Services provided through an interface can be located using a naming service. Based on a name or an identifier such *naming service* returns the necessary information needed to connect to and use the provided services. How the data (arguments and return values) are transfered between the objects that interacts is hidden for the application programmer. The methods of remote services can be called in a similar way that the methods of local objects.

2.1.1 Existing middleware platforms

OMG Common Object Request Broker Architecture (CORBA) was the first middleware platform to gain widespread acceptance [88]. CORBA supports different hosting platforms and different programming languages. All major platforms and languages are supported, and interoperability between different CORBA implementations is a major concern (based on GIOP, and especially IIOP).

Initially CORBA adopted a black-box philosophy with a static programming model, and no guidelines or specifications regarding the implementation issues and access to lower level behavioural details. Recent CORBA specifications (and implementations) include some possibility for customisation and a more open implementation [89]. The Dynamic Invocation Interface (DII) allows a client to directly access the underlying request mechanisms provided by CORBA. Applications use the DII to dynamically issue requests to objects without using the static stubs generated from interface specifications in the Interface Description Language (IDL). DII also supports other types of invocations like non-blocking deferred synchronous method calls (separate send and receive calls) and one-way calls (send-only calls). The Portable Interceptors make it possible to add extra services to the core behaviour of CORBA. More precisely, interceptors can be added as pre- or post-processing activities performed on invocations and replies. Interceptors can also be related to the process of creating objects (to customise this process). Policy objects are another CORBA feature that can be used to control the behaviour of some internal CORBA services. Policy objects can be installed once. Since they can not be overridden they do not provide mechanisms for dynamic behavioural changes.

The Java platform is tied to the Java language but is available for all major host platforms. The Java Remote Method Invocation (RMI) protocol is an object-oriented protocol for performing remote method calls. Access to remote methods are made available through remote interfaces. Recent Java RMI implementations also makes it possible to cross the language barriers. Java RMI using CORBA IIOP as its transport mechanisms has made it possible to call methods available in CORBA interfaces (implemented in any language supported by CORBA) using Java RMI, and vice versa. Java application servers hosting Enterprise JavaBeans (EJB) components also use Java RMI as its communication mechanism. The added services provided include complete life-cycle management of components, transactional component behaviour, and security features (access control). Behavioural issues including life-cycle management, transactional behaviour and access control can be specified in the deployment descriptor of EJB components. When a component is deployed these behavioural features of the component can not be changed.

Microsoft provides its own component model COM [110]. This is a language independent component model providing language independent interactions between components implemented in different languages. COM is closely tided to the Microsoft platforms. To add support for distributed applications an extension to this model named DCOM [84] has been made available. COM+ [69] introduced more advanced features in this model. These features include transaction support and security mechanisms. The most important new feature promoting openness to the model itself is COM+ interceptors. They can be used to change a limited set of aspects of a running system. These aspects includes classes, interfaces and methods. .NET adds the possibility to associate attributes (meta-data) with classes, interfaces and methods [73]. These attributes are introduced at compile-time and they can be inspected (but not modified) at run-time.

To summarise, all the existing platforms presented above only provide a limited set of features (hooks) for the configuration and re-configuration of the platform, and in many cases these features are not available at run-time.

2.1.2 ISO RM-ODP

The ISO Reference Model for Open Distributed Processing (RM-ODP) [60, 61, 62] offers a road-map for the design and implementation of middleware platforms. It differs from existing popular middleware platforms in several important ways. First, it specifies different types of interfaces for continuous media, signals and traditional operational methods (invocations). Secondly, it specifies explicit bindings. Finally, it specifies support for real-time synchronisation and dynamic quality of services management functions.

Continuous media and signals do not fit the traditional concept of operational interfaces. An interface for continuous media interacts in a timely manner to transfer frames representing a period of data. This is very different from operational interfaces with a request to a server followed by some processing of the request at the server followed by a reply possible containing some results. The characteristics of a signal is also very different from the characteristics of an operational invocation.

Most middleware platforms provide an implicit binding between remote objects. It is called implicit since the actual creation (and selection) of the binding is hidden for the user. It happens automatically when the user gets access to the interface of a remote object. Such an implicit binding fits well for most operational interfaces because the protocol takes care of transferring the request and reply safely over the network. This might include retransmission and other time consuming tasks. Such protocols do not work very well for continuous media. If a video is played over the network and a frame is not received at the receiving end at the moment it should be played, it is better to throw away this frame and concentrate on viewing the next frame in a timely manner. This can be achieved by using a binding that fits well for this kind of application. Explicit bindings make it possible for the programmer to explicit select and configure the communication between the objects that interact.

The standard includes five different viewpoints. The technology viewpoint is concerned with the technology needed to realise the system. The engineering viewpoint is concerned with the mechanisms supporting distributed interactions. The computational viewpoint views a system as a composition of objects interacting through well-defined interfaces. The information viewpoint is concerned with the representation and semantics of stored and processed information in the system. Finally, the enterprise viewpoint has a focus on business rules, policies and roles in the system.

Evolution and dynamic reconfiguration are provided by the engineering viewpoint and its engineering viewpoint language. It can be used to configure the components and functions of the platform. In addition, the information viewpoint can describe the configuration of the (running) system. However, no specific support to join these two viewpoints for dynamic adaptation are specified.

2.2 Reflection

The reflection hypothesis introduced by Smith in 1982 [101] states:

> *"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operation and structures."*

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a meta-object protocol (MOP) which defines the services available at the *meta-level* (in contrast of the application and object functionality at the *base-level*). Initially, Smith's work inspired a large body of work to the field of programming language design [67, 108, 2]. Later reflection has been applied to operating systems [113] and, more recently, distributed systems [82].

Open implementation is the motivation for reflective systems. If we hide implementation details (by encapsulation) we also make implementation decisions on behalf of the application. Knowledge available when the object is used (and not when it is implemented) could lead to a more effective and suitable implementation. Open implementation exposes such details and makes it possible to alter the implementation according to current knowledge. However, there should be a principled division between the access to the functionality provided by an object and its implementation. The functionality is available through the base-interface of the object and the implementation is available through its meta-interface.

The split between base-level and meta-level (and possibly meta-meta-level) provides a model for *separation of concern*. At the base-level only base-level functionality is of concern. The meta-level exposes implementation details not of concern when programming at base-level.

Reflection provides a *principled* (as opposed to ad hoc) means of achieving open implementation. It can be used to inspect and adapt the system. By exposing the implementation details it becomes straightforward to inspect the behaviour of the system. This can be useful for debugging and to implement different monitor functions. Adaption makes it possible to change the behaviour of existing features or add new features.

2.2.1 Models of reflection

In *procedural reflection* the self-representation of the system is the system's own implementation. Altering the self-representation also alters the implementation of the system. The self-representation and the system is causal connected.

In *declarative reflection* the self-representation is separated from the implementation of the system. Altering the self-representation does not automatically alters the implementation of the system. The causal connection between the self-representation and the implementation of the system has to be explicit maintained.

Structural reflection in an object-oriented context is concerned with the actual content and structure of an object [108]. This includes the data and implementation of the object (including its methods). For composite models the contained objects and the structure of a composition can be exposed and accessed through structural reflection.

Behavioural reflection is concerned with the underlying system [108]. It exposes the internal aspects of the run-time environment. It includes the activities of the underlying system and the resources required to support these activities.

To model the different aspects of reflection in a system the meta-level can be divided in a number of distinct *meta-models* [90]. The benefit of this approach is the simplified interfaces provided by each meta-model of the meta-level. This separation of concern is important in making the task of managing such meta-levels possible.

2.2.2 Reflective languages and systems

Reflection originated in the programming language community [101]. CLOS [52, 67] is an example of a reflective lisp-based programming language. Reflective features in CLOS and similar languages [78, 39] access its interpreter or virtual machine to expose and manipulate the implementation of the language itself (behavioural reflection).

Standard Java also provides some limited reflective features [102]. These features can only be used to inspect the structure of language entities (attributes and methods) and then use this information to operate on these entities (limited structural reflection).

The Python language [76] also includes some reflective features [6]. These features can be used to access and manipulate its language entities (structural reflection). Some limited access (hooks) to the underlying system is also available (limited behavioural reflection). One example is the meta-class hook that can be used to alter the behaviour of the constructor of a class.

Several research projects have extended existing languages with reflective features. Examples are OpenC++ [38], Iguana [54] and Reflective Java [111, 112].

Reflection has also been introduced in other type of systems with the need for customisation and dynamic adaptation. Examples are the operating system Apertos/Aperios [113] and (distributed) architectures like CodA [81, 82] and AL-1/D [90].

2.2.3 Reflective middleware

Adding reflection to middleware to support customisation and run-time adaptation has been done in several projects. Open-ORB [28] will be introduced in great details in the next chapter. The following will list some other related projects and platforms.

The middleware platforms presented in Section 2.1.1 above only provide limited or no access to implementation details or the underlying structure of the system. CORBA has recently added features like dynamic invocation interfaces, portable interceptors and policy objects, Java provides some limited inspection facilities and (static) deployment descriptors for behavioural issues concerning EJB components, and COM+ introduces the interceptor concept. However, none of these platforms can be considered to be reflective.

The Multe-ORB [47] is a flexible and adaptable middleware platform with the goal of supporting a broad range of quality of service requirements of distributed multimedia applications. To achieve this reflective techniques are used at the (transport) protocol level.

FlexiNet [59] is a configurable middleware platform with focus on mobility (mobile objects). It provides an open binding framework with configurable protocol stacks. Interceptors can be plugged into these protocol stacks to implement and customise non-functional properties. Examples of such properties include transactional behaviour, access control and replication. FlexiNet is implemented using the reflective extensions of Reflective Java [111, 112]. FlexiBind [57, 58] is an extension of FlexiNet using policies to control the configuration of the open binding framework. Meta-policies can be used to manage run-time selection of policies. This makes it possible to do dynamic adaptation of the entire binding configuration.

In dynamicTAO [95, 70] a CORBA platform is extended with the ability to dynamically add strategies to and remove strategies from existing components. Different configurator classes are available for the management of different entities. The interfaces of these configurators are the meta object protocol (MOP) of dynamicTAO. They are used to load new strategies and inspect components.

UCI/LegORB [96] is a related project using an explicit component based approach. The configurable entities are components that represent customisation slots for different services. These (abstract) components can also be added or removed from the system itself. Example of usage is to provide different personalities of the core system (for example CORBA or Java RMI). Simple personalities with small footprints can be used in hosts with limited resources (PDA) [97].

A similar approach is found in Quarterware [100]. Quarterware is a customisable middleware architecture. In [100] its flexibility is demonstrated by deriving implementations for core facilities of different existing middleware platforms including CORBA and RMI.

mChaRM [35] uses reflection to extend the traditional communication semantics of communication channels with new behaviours and new semantics. A new reflective model called *multi-channel reification* [5] is used for designing and developing complex communication mechanisms.

All these platforms use reflection features to address only a limited set of the aspects of the environment provided by the middleware platform. They either have a focus on adaptation at the network protocol layer or on adaptation on the component structure of the system.

2.3 Quality of service management

It is possible to distinguish between the functional and the non-functional properties of a service. The *functional properties* describes the service provided while the *non-functional properties* describes how this service is provided. A news-on-demands service provides the news as a video-flow. The functional properties of this service is to select, play, pause and stop such video-flows. The non-functional properties are the quality of the received video service (number of colours, audio quality, response time on the controls, and so on). These non-functional properties are the *quality of service* (QoS) parameters.

2.3.1 Quality of Service

The quality of a service says something about how well the service function is performed. This includes something more than just 'provided' or 'not-provided. The *quantification* of the quality of a service differs on different level of abstractions.

The top-level abstraction is usually considered to be the impression the user gets from the quality of the presented service. This can vary depending on the expectations and needs of the user. Using a telephone the user expects the audio (the spoken words) to be easy to understand and without any notable delays. This makes a phone conversation (interaction) work as expected. A Hi-Fi enthusiast who is investing a lot of money and time in her HW (amplifier, speakers, turntable, and so on) and SW (CDs and other sources) expects the sound quality to be as close as possible to a live concert. Delay and other non-audio-quality properties are of less importance. Quality properties, like the impression a user has of the quality of the audio in a given service, can only be measured *subjectively*. For example, a user can rank the quality of a given service using the terms 'bad', 'reasonable', 'good', 'very good' and 'excellent'.

At a lower level of abstraction the quality of a network connection can be measured in throughput (bits per second), delay (milliseconds), packet-loss (percentage), smoothness

QoS characteristic	Example value
User impression of phone audio quality	'good'
Network throughput (bandwidth)	1.8 Mbit/s
Maximum delay	23 ms
Maximum packet loss	1 %
Smoothness (variation of delay)	$\pm$3 ms

Table 2.1 Examples of QoS characteristics.

(throughput variations) and so on. These quality properties are easy to quantify with numbers. The throughput of a given network connection can be 1.87 megabits per second with an average delay of 23 milliseconds. Quality properties like these are measured *objectively*.

A given level of abstraction is based on QoS provided by lower level of abstractions and its QoS abstraction is provided to higher level of abstractions. For example, a media player has a level of QoS abstraction that specify the synchronisation between an audio stream and a video stream. These values say something about two streams with given qualities and how much these two streams can be out of sync and still provide the specified (lip-sync) quality. The quality of each stream is provided by a lower level of abstraction, and the provided abstraction is used at a higher level to specify the user level quality (the user impression).

In the RM-ODP document "QoS – Basic Framework" [63] quality of service is defined as "a set of qualities related to the collective behaviour of one or more objects". The term "collective behaviour" gives a hint about the different abstraction levels. At a higher level the observed quality of a service is based on the collective behaviour of lower level functionally and how well they perform according to the expected quality. The observed audio quality of a phone application is based on the lower level quality of the network connection including the network throughput, delay, packet loss and smoothness. The resulting perceived quality for the user determines if the lower lever QoS characteristics are sufficient in the given setting.

Quality of service (QoS) is expressed as *QoS characteristics*. The QoS *requirements* and *capabilities* represents what the user (or a component) wants and what the system provides, respectively. A system can meet the requirements of a given application. Examples of some QoS characteristics at different abstraction levels are provided in table 2.1.

2.3.2 Management

QoS management is the task of establishing and maintaining these non-functional properties. As stated in the previous chapter, this work will focus on the *dynamic quality* *of service management* functions. In particular, the focus will be on monitoring and maintenance of quality of service properties. *Monitoring* is concerned with monitoring the quality of the service being offered and reporting any problems of achieving this. *Maintenance* is concerned with actions that can be taken to sustain the quality of service.

Static QoS management functions are ignored in this work for two different reasons. The first reason is that a lot of work on static QoS management functions has already been presented in the literature (examples of such work can be found in [55, 33, 92, 74, 86]). An overview presenting a lot of interesting work on this topic is found in [15]. The other reason is that the adaptation mechanisms needed for dynamic QoS management functions match the features provided by a reflective system very well. In [107] the use of reflective features to implement a CORBA Component Model compliant ORB that dynamically adjusts component QoS properties at run-time is explored.

The QoS characteristics that the dynamic management system should manage are expressed using a QoS *specification* [1]. The specification provides a description of the quality the management function should help the system to satisfy. A QoS specification at a higher level has to be mapped down to a set of low level specifications (in combination with other higher level specifications). This work will not include details about QoS specifications and the mapping of higher level specifications to a set of lower level specifications.

Monitoring is used to collect the information needed in the management task. A static set of monitors are not flexible enough in a reflective system that can be re-configured at run-time. Interceptors and other reflective features can be used to dynamically introduce monitoring into a running system [109, 70]. Monitors should also filter the collected information and only forwards the relevant information to the decision process.

To maintain the specified QoS the management system has to perform the management functions based on the input from its monitors (and the state of the management functions). These management functions includes the decision making process (what actions, or any actions?), and the process of performing the decided actions. Reflective systems provide the possibility to adapt. These adaptation functions support a flexible set of management functions (performed maintenance actions) [71, 72].

Chapter 3

Open-ORB

This chapter gives an introduction to the Open-ORB[1] architecture at Lancaster University. Open-ORB is a reflective middleware platform supporting inspection and adaption of application and system components. The work described in details in part II (OOPP) is closely related to the Open-ORB project.

3.1 Reflective middleware

The role of middleware is to present a unified programming model to application writers and mask out the problems of heterogeneity and distribution [16, 45, 68]. Middleware has a similar role in a distributed system that the operating system has on a computer: to hide the low level details and present a unified programming model. Middleware and operating systems also share common problem related to their usage. The problem was stated clearly by Per Brinch Hansen in his book "Operating System Principles" [34]:

> *"One of the difficulties of operating systems is the highly unpredictable nature of the demands made upon them."*

Middleware has to remain responsive to new challenges and demands from existing and new type of applications. Some of these new difficulties that are emerging are

i) support for multimedia,

ii) real-time requirements, and

iii) increasingly mobility.

These different and unpredictable challenges are the motivation for the next generation middleware architecture in Open-ORB [28]. One key feature to meet these challenges is adaption. Randy H. Katz summarises the requirement for adaption with mobility in the following observation [66]:

> *"Mobility requires adaptability. By this we mean that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion."*

It should be possible to configure the underlying support offered by the middleware platform to satisfy the requirements from a wide variety of applications. Example of such configurations are scheduling policies, special protocols for multimedia and resource management. Another important requirement is the possibility to inspect and adapt the support offered at run-time. This is in Open-ORB done by adapting an open engineering approach through the concept of reflection (see section 2.2).

Current generation middleware only have limited (if any) support for configurability and open engineering. Implementation details are hidden and services are available through a set of interfaces (APIs) to a black box. There are several good reasons for doing this[2] but recent experiences with these platforms suggest that this is to restrictive to a lot of application types. OMG has as a result of this recently added some interfaces to the underlying system i CORBA. But these approaches provide limited openness to a limited set of selected components and they are rather ad hoc.

3.2 General principles

The Open-ORB architecture tries to overcome the limitations in current middleware platforms by opening up the ORB. This is done through the concept of reflection. The general principles of this architecture is described below.

3.2.1 Procedural reflection

A reflective architecture can either support procedural or declarative reflection [78]. In the declarative approach the self representation is given by a set of declarative statements describing the behaviour of the system. This abstract approach focuses more on what the implementation of the system should achieve and not on how this is achieved. The casual connection is however more difficult to achieve in a declarative approach since the behaviour of the system is not observed (and accessed) directly. In the procedural approach the representation of the system is given by the actual implementation. The representation of the system then shares the computational model of the system. The representation of the system available through reflection is called the meta-space of the system. Programming done in the meta-space is called meta-space programming.

Another advantage of a procedural approach is that it is more primitive and has less restrictions compared to a declarative approach. It is also possible to build a declarative interface on top of procedural reflection if a more abstract model is convenient (but not vice versa). An procedural approach to reflection is chosen in the Open-ORB architecture.

3.2.2 Object-oriented

Object-oriented models have a predominance in open distributed processing systems [29, 87] and Open-ORB adapts such a model of computation.

The object-oriented model is common in reflective languages and systems. The important synergy between reflection and object-orientation explained by Kiczales et al in "The Art of the Metaobject Protocol" [67] is an important motivation for this:

> *"Reflective techniques make it possible to open up a language's implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language's implementation and behaviour to be locally and incrementally adjusted."*

3.2.3 Per object meta-spaces

Since an object-oriented programming model is adopted, a per object way of applying reflection is also adopted. The result is a fine level of control over the support provided through the meta-space. The limit of the scope of change also minimise the problem of maintaining the integrity of the system. Changes done through reflection of one object does not directly effect the other objects. A heterogeneous environment includes different types of objects with a variety of capacities for reflection. This difference also encourage per object meta-space. In situations where it is useful to be able to access the meta-space of sets of objects in a single operation the intention is to use a group mechanism. ABCL/R2 [79] provides an example of how such a group mechanism could be implemented.

3.2.4 Multi model approach

Like in the AL-1/D framework [90] the meta-space is divided in several meta-models represented by meta-objects. Each object have potentially one meta-object per meta-model. The behaviour of an object can be changed by modifying its meta-space through its meta-objects. The meta-objects are causally connected to their (base-level) object. The meaning of this is that changes done to an object through its meta-object are immediately visible in the object (and vice versa).

The motivation for multiple meta-models is separation of concerns. Each model provides access to distinct structural or behavioural issues of an object. Structural reflection is concerned with the actual contents of a given object and behavioural reflection is concerned with the activity in the underlying system [108]. Four distinct meta-models have been identified and developed in the current Open-ORB architecture. A closer description of each model follows in section 3.4.

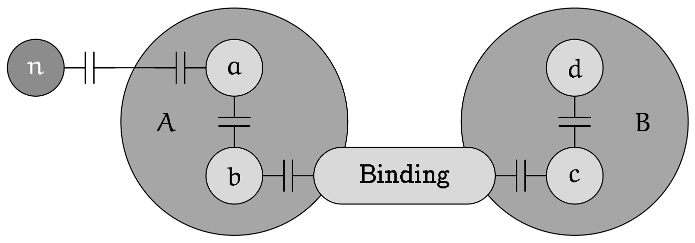


Figure 3.1 Components a, b, c, d and an explicit binding with their infrastructure.

3.2.5 Other principles

It is a strong level of recursion in the meta-object approach in Open-ORB since meta-objects are, like other objects, also open to reflection. And since a meta-object only exists theoretically until it is actually accessed infinite levels of meta-objects are (theoretically) available. This is similar to the reflective approach in ABCL/R [108]. Such access to different meta levels is important since it makes it possible to inspect, manipulate and extend even the meta-models (and meta-meta-models) of objects and interfaces. This can be used to expose and modify the policy behind the adaption provided.

3.3 Programming model

The programming model adapted in this work is mainly influenced by the RM-ODP [60, 61] computational viewpoint where

i) objects can have multiple interfaces,

ii) operational, stream and signal interfaces and bindings are supported, and

iii) explicit bindings can be created between compatible interfaces.

Many of the concepts introduced here match the definitions found in the foundations and the architecture part of the RM-ODP documents [61, 62].

Objects and bindings are contained in capsules (managed address spaces). Remote bindings can make connections between interfaces in different (remotely separated) capsules. Interfaces and capsules can be located using a naming service. Figure 3.1 is an example of some components and their infrastructure. A and B are two different capsules containing the components a, b, c and d. Object a is bound to a name server n with an implicit binding. There is an explicit binding between object b and c. The other interfaces of the objects and the bindings are connected with local bindings.

The programming model presented here does not include the concept of clusters from RM-ODP. Clusters in RM-ODP are the unit of deactivation, check pointing, reactivation, recovery and migration [62]. The unit for these functions in Open-ORB are components (see 3.3.2) and abstract resources (see resource meta-model in section 3.4.4).

3.3.1 Interfaces and local bindings

Interfaces represent access points to objects (and components). Each interface represents a set of methods related to its object. Interfaces are introduced to disconnect statically relations between objects. An object with interfaces can connect (bind) to any other object with a matching interface. The methods of an object are then called through their interfaces and not directly using the object and method name pair. This includes method calls to the object and from the object (incoming and outgoing method calls or messages).

When two objects with matching interfaces have been created, there is no association between them until they are explicitly bound to each other. The most primitive form of binding that can be created between two interfaces is a local binding. A local binding is always between interfaces in the same address space (capsule). When a local binding between two interfaces is created type checking is performed to see if they match. The type (or the signature) of the interfaces can be described in CORBA IDL or a similar syntax.

3.3.2 Components

Objects usually exists as components in the Open-ORB framework. A component is a unit for composition and independent deployment [103]. They are developed and delivered independently and provide access to its requested and provided services (methods) through one or more specified interfaces. All interactions between components are specified through these well defined interfaces. Such an interface connection architecture has a basic conformance criteria that says that the system's components interacts only as specified in their interfaces [75]. The connections (the dependence between components) are explicit and an attempt to locate components affected by changes to an interface becomes easier. This is important when possible replacements and restructuring of components in a system are determined.

Components take the interface approach one step further. It makes the interfaces of an object available without any pre-known knowledge of the object providing them. The interfaces provided can be browsed and access to a given interface is done by its key (name). A primitive component encapsulates one object and provides access to and from the object through one or more interfaces.

A composite component contains a set of other components represented with a component graph. Composite components makes it possible to represent a complex structure of components bound together in an object graph as a single component with a set of well defined interfaces.

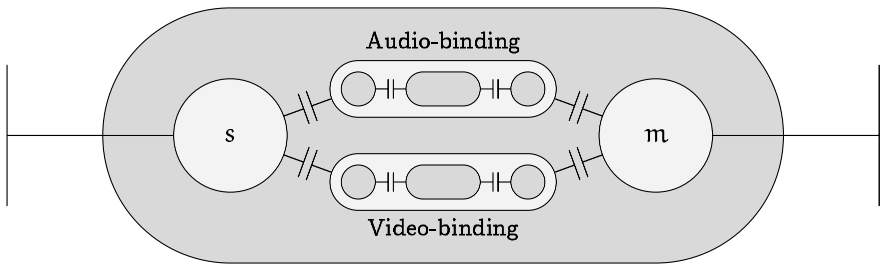


Figure 3.2 The AV-stream is a composite component containing a split component s, an audio-binding, a video-binding and a synchronisation component m. The audio- and video-binding are composite components.

The component graph specifies the local bindings between the different interfaces of the components contained in the composite component. A component contained in a composite component can again be a composite component. This recursive component containment is terminated by primitive components. The set of external interfaces provided by a composite component is a mapping from a subset of the interfaces of its contained components.

An example is the AV-binding (audio-video binding) component in Figure 3.2. It contains a split component s (splits the AV-stream in one audio stream and one video stream at the sending side), an audio-binding component, a video-binding component and a synchronising component m (merges and synchronises the audio and video stream at the receiving side). The split and synchronising components are primitive components, but the audio- and video-binding components are composite components containing stubs and lower level bindings. The layered structuring is terminated in audio- and video-binding components if their contained stubs and lower level bindings are primitive components.

A composite component can be distributed. A distributed composite component contains components located in different address spaces. These address spaces can be located on different nodes. Components in different address spaces can be bound (connected) with binding objects.

Binding objects hide the complexity and protocol issues of remote communication and help providing location transparency. A binding object is a component, and usually a distributed component. A typical binding object is a composite component containing stubs and a lower level binding.

3.3.3 Capsules

Components in Open-ORB exist in managed address spaces called capsules. A capsule provides a set of services for the managing of the components located locally. Every component in the Open-ORB programming model is under the control of one capsule. The services of the capsule can be provided both to local and remote components.

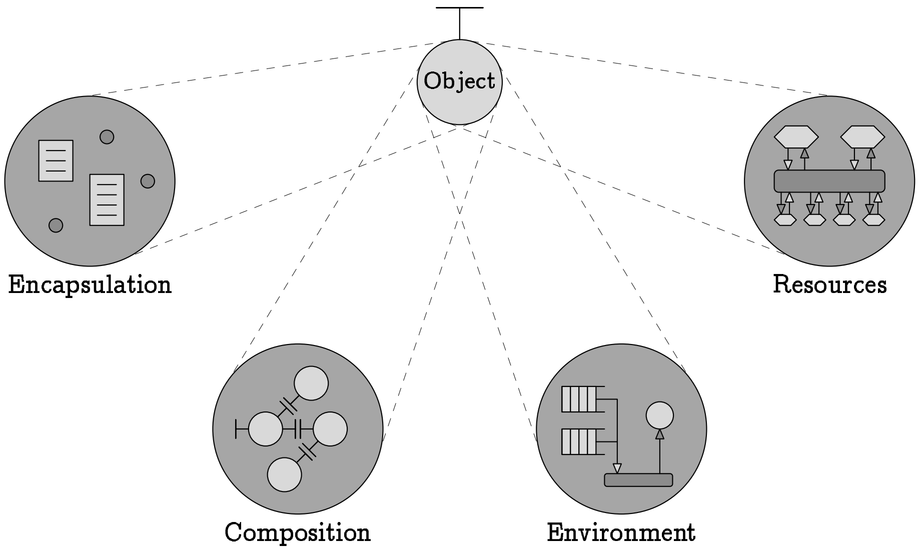


Figure 3.3 An object and its four meta-models.

3.4 Reflection through meta-models

We can distinguish between structural and behavioural aspects of reflection [108]. The structural aspects is concerned with the actual contents of a given object and the behavioural aspects are concerned with the activity in the underlying system. In Open-ORB these two aspects are represented by four distinct meta-models [90]. Structural aspects are represented by the encapsulation and the composition meta-models, and behavioural aspects are represented by the environment and the resource meta-models. Figure 3.3 illustrates an object with its four meta-models. Each meta-model of the object are accessed through its meta-object [108]. A description of each model follows below. The meta-models representing the behavioural aspects are only given a brief description since the exploration of them still is ongoing work. The prototype presented in the following chapters does not implement these meta-models.

3.4.1 Encapsulation meta-model

The encapsulation meta-model opens up the encapsulation provided by objects. It makes it possible to inspect, modify and extend the implementation of an object. This meta-model can be used to monitor and control all access to an object (including access to its attributes and methods).

The encapsulation function returns an encapsulation meta-object representing the encapsulation meta-model of the given object or interface. The encapsulation meta-object is always located in the same capsule as the object or interface it is controlling. Access from another capsule could be provided by a meta-object proxy with a implicit binding to the actual meta-object.

Typical operations done on a component or an interface using the encapsulation meta-object include to add a new method to the component, to add pre-, post-, and wrapping-methods to a method of the component and to replace the implementation of a method of the component.

3.4.2 Composition meta-model

Composite components are used to group together a set of components that naturally belongs together. They also ease the handling of complex components through its layered structuring terminated by primitive components.

Typical composite components are complex binding objects. The adaptability needed for multimedia applications in a mobile environment identifies a need to manipulate and restructure (change and extend) the component graph of such complex binding objects during their life cycle. A composition meta-model is provided to achieve this.

The composition meta-model is empty for all non-composite components (primitive components). The composition meta-model provides access to the component graph representing the composite component. The composition function returns the composition meta-object representing the composition meta-model of the given component. Every external interface of the composite component return the same composition meta-object. The components found in the component graph of a composite component are not necessary located in the same capsule, but they are all accessible from the same composition meta-object. Typical operations done on a composite component through its composition meta-model are replacement of contained components and extension of the existing component graph with new components [51]. An example is to add a buffer component to the sink side of a stream binding to handle jitter.

3.4.3 Environment meta-model

The execution environment of interfaces manages messages (method calls) sent through these interfaces. The management includes synchronisation and scheduling between messages sent and received. Access to the execution environment can be used to change the policy of this management. This management in ATOM [91] is separated from the actual implementation of the methods.

The environment meta-model exposes the execution environment of each interface. This includes on the sender side of a method call (or message) initiating, enqueing, selection, marshalling and transferring, and on the receiver side of a method call arrival, enqueing, selection, unmarshalling and dispatching. An example of a possible implementation of this model is described in [42].

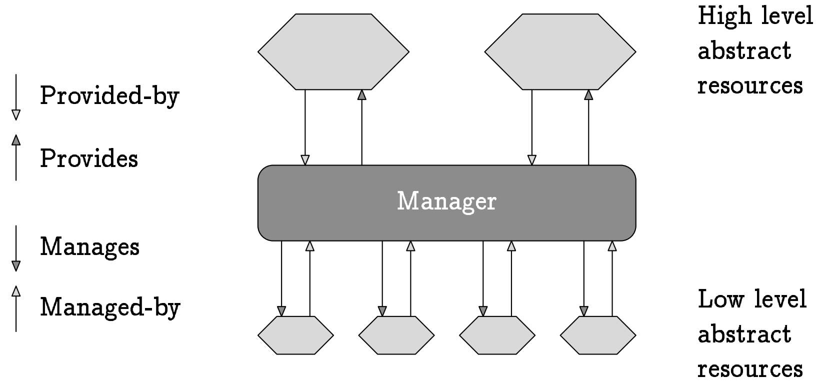


Figure 3.4 The resource framework consists of managers managing lower level abstract resources and providing higher level abstract resources.

The implementation of this model in Open-ORB is a candidate for further research, but related work has been done in ABCL/R [108] and CodA [82]. The environment function will return an environment meta-object that is a composite component representing the environment of the given interface. The object graph of this composite component can be manipulated by fetching the composite meta-object of the environment meta-object.

3.4.4 Resource meta-model

Multimedia components like continuous media objects require a guarantee for available resources to perform as expected. The resource meta-model exposes the allocation and management of (abstract) resources associated with an object or an interface. These resources are a part of the resource framework that provides a complete and consistent model of the system resources at different levels of abstractions. Higher level of abstract resources are provided by managers managing lower level of abstract resources [94]. Figure 3.4 illustrates the relations between a manager and abstract resources (see [25] for more details). The resource function returns a resource meta-object used to manipulate these abstract resources.

Notes

1. More information about the Open-ORB project is available from http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/

2. The main reason to provide a middleware platform as a black box with a set of pre-defined interfaces is to hide the complexity of the underlying system for the application programmer. For a lot of traditional client-server based applications is this hiding of complexity (transparency) a good thing.

Part II

Essence

Chapter 4

OOPP design issues

This chapter will present some design issues for OOPP, the prototype of the Open-ORB architecture described in the previous chapter. The presentation of these issues are split into three sections. The first describes the selected programming model, the second describes the infrastructure for this programming model, and the last describes the meta-models. But before that, a brief description about the influence from the RM-ODP framework will be presented.

4.1 RM-ODP

The programming model and the infrastructures of OOPP are influenced by the ISO Reference Model for Open Distributed Processing (RM-ODP) [60]. RM-ODP is an object-oriented system specification methodology based on the concepts of viewpoints. It provides vocabulary and grammar for describing a distributed system from each viewpoint. OOPP does not have a similar viewpoint concept. However, the separation between a base-model (the programming model and its infrastructure) and a set of meta-models in OOPP is related to the viewpoints of RM-ODP.

The computational viewpoint of RM-ODP is concerned with the description of the system as a set of objects (or components) that interacts. This interaction happens at interfaces. This is closely related to the programming model provided by OOPP. In OOPP a system contains a set of objects or components that interacts through interfaces. Interfaces are connected with local bindings or with binding objects.

The engineering viewpoint of RM-ODP is concerned with the mechanisms supporting system distribution. This is closely related with the infrastructure of OOPP. The infrastructure of OOPP provides capsules similar to capsules in RM-ODP. The capsules are managed address spaces (storage) with processing resources. The infrastructure of OOPP also consists of name servers, node managers and factories.

The technology viewpoint of RM-ODP is the closest match to the reflective features of OOPP. However, the reflective features of OOPP available through the different meta-models provides a completely different approach to accessing the implementation of the system. The meta-models are available for objects and components representing both the base-level (application) and the infrastructure.

One important reason to use (a limited set of) the rich vocabulary and grammar of RM-ODP can be summarised in two features:

i) Different types of interfaces for operational methods, continuous media and signals

ii) Explicit bindings

These features in OOPP will be elaborated below.

4.2 The programming model

The OOPP programming model provides interfaces, components and bindings as its key features. Similar features are found in the computational viewpoint of RM-ODP [61].

4.2.1 Objects and classes

In OOPP the term object have its traditional meaning. More precisely, an object in OOPP is a Python *object*. The implementation of an object is given by its *class*. An object is an instance of its class created with the constructor of the class.

4.2.2 Interfaces and local bindings

An *interface* of an object defines a subset of the interactions of that object [61]. Examples of such interactions are invocation of methods provided by this object and invocations of methods in other objects done from this object. It is important that these interactions includes both in-coming and out-going method calls since every interaction of the object should (possibly) be defined by interfaces. If every interaction of an object is defined by a set of interfaces, this set of interfaces represents a complete description of every possible interaction of that object.

It is possible to distinguish between different types of interfaces. Interfaces of ordinary method invocations are named *operational* interfaces. Other possible interface types includes *signal* and *stream* interfaces. The need for different interface types is motivated by the different behaviour and characteristics of these types of interactions. A signal contains none (or at least a small amount of) data and should be safely delivered in-time to the receiver. A stream is a sequence of data frames that should be transfered in a periodical cycle possible with real-time constraints.

An operational interface could possibly contain methods provided and methods requested. Methods provided by an object through an interface are called *exported methods* of that interface. Methods requested by an object through an interface are called the *imported methods* of that interface. It is possible to decide to have two different types of operational interfaces. One type used for exported methods and one type used for imported methods. An object could then be using interfaces for exported methods to provide its methods to other objects and interfaces for imported methods to access the methods of other objects.

Interfaces are illustrated in the following simple example. An object a implements a method f and have an interface i. Method f is available for other objects through interface i since interface i exports f. An object b has an interface j that imports a method f. Object b uses interface j to call an external method f implemented in an object with an interface that exports a method f. This other object could be object a with interface i.

OOPP does not distinguish between operational interfaces that export and import methods. Every operational interface in OOPP can contain both exported and imported methods. The programmer however is free to adopt a style of programming where she distinguish between interfaces for exported and imported methods. The main motivation behind two-way operational interfaces in OOPP is the flexibility of this approach. This approach does not enforce any particular style of programming. An application programmer can decide to use only one-way operational interfaces (operational interfaces with either only exported methods or only imported methods). She could even create her own specialised interface class with behaviour based on the original two-way operational interface class. In some cases, the interaction between two objects could at one point be initiated from one object and later be initiated from the other object. It is possible with two-way operational interfaces to implement these interactions with one interface at each object. An example is an object a that call a method of object b to register for a given event. Later, object b calls a method of object a to notify object a that the event occurred. These interactions can be implemented using one two-way operational interface in both objects.

The methods provided by an object are exported in one or more interfaces. The methods of other objects used from the object are imported in one or more interfaces. One reason to introduce interfaces is to disconnect a statically relation (binding) between objects. An object with an interface can connect (bind) to any other object with a matching interface. The method calls are not done directly to the object but through its interfaces.

When two objects with matching interfaces have been created, there is no association between them until they are (explicitly) bound to each other. This means that object b can not call method f in interface i of object a through its interface j before a binding is created between interface j and interface i. The most primitive form of binding that can be created between two interfaces is a local binding. A *local binding* is a way of saying "the reference to the imported method f in this interface is actual the exported method f of this other interface". Figure 4.1 illustrates a local binding between interface i and j of object a and b. Interface i exports the method f of object a and interface j imports a method f. Object b can call method f of object a when interface i and j is bound with a local binding.

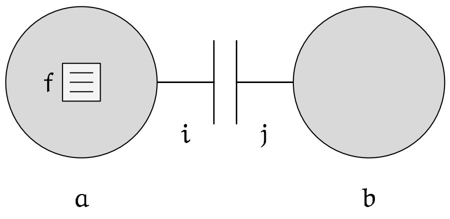


Figure 4.1 Two objects a and b with interfaces i and j bound with a local binding.

A local binding is always between interfaces in the same address space (capsule). When a local binding between two interfaces is created type checking is performed to see if they match. The type (or the signature) of the interfaces could be described in a CORBA IDL like interface syntax. Current version of OOPP only checks if the names of the methods match. Two interfaces match when the exported methods of one interface match the imported methods of the other interface (and vice versa).[1]

A local binding between two interfaces is the mechanism provided to connect (bind) and disconnect (break) matching interfaces. An interface can be bound with a local binding to any matching non-bound interface in the same address space. A local binding can later be broken. The involved interfaces are then unbound and can be reused in another local binding.

An interface is only useful together with another interface. If an object a has an interface i with an exported method f it is not possible to call this method directly through interface i. Another interface, for instance j, that imports a matching method f has to be used to call the method. When a binding is created between these two interfaces method f can be called through the interface j in the same way it can be called directly through object a. This makes it possible to use an interface as a proxy (in CORBA terminology) for an object. Typically, such an interface j imports every method of a, and is bound to an interface i of a that exports every method of a. Interface j can now be used to call the methods of a in the same way as if it was a itself. This is particularly useful when the objects are remote to each other. The local binding can transparently be replaced with a binding over a network (see bindings in section 4.2.4).

The only way to access an interface is through its *interface reference*. An interface reference can be (with support from the infrastructure) a global identifier for an interface. It is the unit used to export and import services in the programming model. An interface reference contains the description of the exported and imported methods and a reference to the object the interface is related to (possible none if the interface only imports methods). By convenience, an interface reference and the interface it is representing are often referred to using the same name.

Stream and signal interfaces exist in a *source* and a *sink* pair. A matching stream or signal interface source and sink pair can be bound with a local binding. The signal or stream from a source interface is transfered to the sink interface it is bound to.

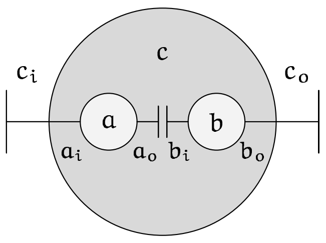


Figure 4.2 A composite component c containing two components a and b.

4.2.3 Components and composite components

Components are the building blocks of the OOPP programming model. A *component* in OOPP is an object exposing its interfaces in a standard way or a special object exposing the interfaces of the object (or the objects) it wraps in a standard way. Each interface of a component is represented by a mapping from a name to an interface reference.

A *primitive component* encapsulates one object and provides access to (exports) a set of its methods through one or more interfaces. These interfaces also provide access to (import) external methods used from the encapsulated object. A component class is provided to create components encapsulating one object.

A *composite component* contains a set of other components represented with a component graph. Composite components represents a possibly complex structure of components as a single component with a set of well defined external interfaces. A composite component always contains components and not just ordinary objects. The reason is that the component graph describes the connections between the interfaces of the contained components, and these interfaces have to be exposed in the standard way provided by components.

The *component graph* specifies the local bindings between the different internal interfaces of the components contained in the composite component. A component contained in a composite component can again be a composite component. This recursive component containment is terminated by primitive components.

The set of interfaces provided by a composite component are a mapping from a subset of the interfaces of its contained components. This implies that an interface provided by a composite component is a mapping to an interface in one of the contained components. Figure 4.2 shows a simple composite component c with two (external) interfaces c_i and c_o, two contained components a and b and one local binding between interface a_o and b_i in the object graph. The external interface c_i is a mapping to interface a_i of component a and the external interface c_o is a mapping to interface b_o of component b. A composite component class is provided to create composite components from a set of existing components.

A composite component can be distributed. A distributed composite component contains components located in different address spaces (capsules). These address spaces can again

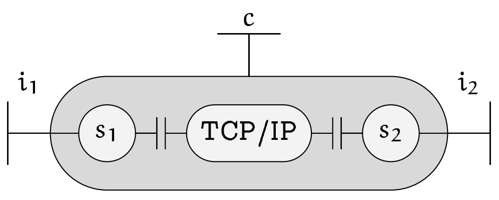


Figure 4.3 A binding containing two stubs s_1 and s_2, a TCP/IP binding and three interfaces i_1, i_2 and c. Interface c is a control interface.

be located on different nodes. Components in different address spaces can be bound with a special type of components called bindings.

4.2.4 Bindings

Connections (communication) between components in different address spaces possibly on different nodes is enabled with a special component type called *bindings*. A binding hides the complexity and protocol issues of remote communication and helps providing location transparency.

A binding is a component, and usually a distributed component. A typical binding is a composite component containing stubs and a lower level binding. Figure 4.3 is an example of a binding object containing two stubs s_1 and s_2, and a TCP/IP binding. Most bindings have (at least) three interfaces, two for the actual binding between two interfaces (i_1 and i_2 in Figure 4.3) and one to control the binding (c in Figure 4.3).

A binding can either be implicit or explicit. An *implicit binding* between two interfaces is created by the infrastructure and is under no control of the application programmer. An implicit binding is usually created when a service is imported. An *explicit binding* is based on the programming structures used by the rest of the application components in the programming model. The creation is usually initiated and controlled by the application programmer. This gives the application programmer control over the creation and configuration of the binding. This is important in order to support multimedia and meet the demands from new application areas.

Three different kinds of bindings are available in OOPP:

i) Operational bindings

ii) Signal bindings

iii) Stream bindings

An *operational binding* is used to forward a method call from an interface importing (requesting) the method to another interface exporting (providing) the method, and (optionally) returning the result back. This kind of binding can either be generic supporting

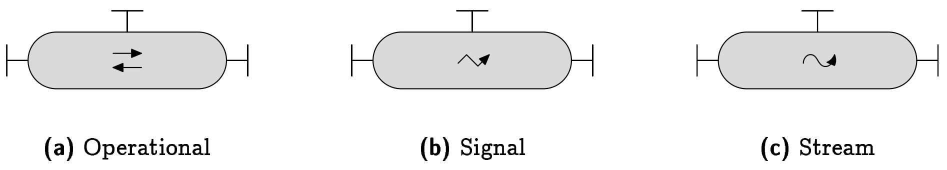


Figure 4.4 Different kinds of bindings used to connect operational, signal and stream interfaces.

operational interfaces with arbitrary signatures, or it can be specialised to support only one type of interface pairs. Generic bindings adapt to any interfaces with any signatures. A generic binding can be compared with the Dynamic Invocation Interface (DII) of CORBA. However in CORBA this is done by directly accessing the underlying request mechanisms provided by an ORB. In OOPP the generic bindings adapts to the given interfaces and the interface-specific stubs are not bypassed. A specialised binding supports only interfaces with a given signature.

A *signal binding* supports a set of one-way signals (communication paths) where no replies are possible. A signal flows from a signal source (sender) to a signal sink (receiver). A *stream binding* supports continuous media like audio and video. A stream flows from a stream source to a stream sink. Figure 4.4 illustrates how these different kinds of bindings usually are drawn.

A standard operational binding is provided. It can be used in the same way as a local binding between two interfaces with the exception that the interfaces can be located in different address spaces (capsules) and/or on different nodes. Since standard operational interfaces can have both exported and imported methods this standard binding provides a two-way operational binding. A two-way operational binding supports method calls initiated in both directions (interfaces on both sides of the binding can export and import methods). This standard operational binding is created with a provided operational binding class. The interface references of the two interfaces to be bound are provided as arguments to the constructor of this class. The result is a binding with interfaces that matches the two given interfaces. This is a generic binding since it will adapt to any interface references provided as arguments to the constructor.

Three different versions of operational bindings are provided [62]:

i) Synchronous interrogation binding

ii) Asynchronous interrogation binding

iii) Announcement binding

A *synchronous interrogation binding* (or an RPC binding) supports method calls with a call (invocation) and reply (termination) structure. The call is blocked until the reply is received from the invoked method. An *asynchronous interrogation binding* is similar to the synchronous interrogation binding, but the call is not blocked. The caller must later invoke a receive reply call to pick up the result (reply). An *announcement binding* supports only the invocation part of a method call. No reply is possible.

A traditional operational binding (for instance in RM-ODP) has one client stub and one server stub. Since the standard operational binding provided in OOPP is a two-way operational binding, both stubs are equal and have the functionality of a client stub and a server stub. Implicit bindings provided by the infrastructure are often synchronous interrogation bindings since their characteristics can be compared with local bindings.

Access transparency hides the difference between accessing local and remote services. Implicit bindings and a matching interface or proxy are used in OOPP to create access transparency. The proxy is used to access the remote interface in the same way as a local interface is accessed. The result of the import services of the name server (see Section 4.3.2) are examples of such proxies providing access transparency.

4.3 The infrastructure

The infrastructure is the supporting environment for programs in OOPP. A component becomes a part of this infrastructure when it is located in a managed address space called a capsule. The infrastructure is influenced by some parts of the engineering viewpoint of ISO RM-ODP [62].

4.3.1 Capsule

A managed address space in OOPP is called a capsule. A *capsule* provides services for its local components (the components located in the address space it manages). It can also provide services to remote components through a capsule proxy. A capsule proxy and a local capsule have identical interfaces, but the requests through a capsule proxy are forwarded to the capsule it represents and the replies are returned back to the caller through the proxy.

The purpose of a capsule is to provide an environment for the OOPP components to exist in. An OOPP component is *deployed* in a given capsule. It can later be destroyed or migrated to other capsules. A component that is deployed in a capsule is instantiated in the address space managed by the capsule and registered in the component register of that capsule. Every registered component is given a local unique identifier. A local unique identifier is guaranteed unique in its capsule. A global unique identification of a component is constructed from a capsule proxy representing its capsule and its local unique identifier.

The capsule provides services to get the *global interface reference* of an interface of a registered component by using the unique local identifier of the component together with the name of the interface. The returned global interface reference differs from the (local) interface references discussed earlier in one important way. They provide a global unique identification of an interface. They contain enough information to be used in any capsule still identifying the right interface, even if this interface is located in another remote capsule. The differences between a local and a global interface reference are usually hidden for the application programmer. However, the application programmer should always use interface references provided by the capsule (or a naming service) when she creates distributed applications (applications involving different capsules).

The reason local interface references exist at all is related to when different parts of a complex environment like OOPP come into existence. The programming model presented in section 4.2 exists before (or without) the OOPP infrastructure. An application programmer creates interfaces using local interface references without the knowledge of the infrastructure. The infrastructure later automatically transforms these local interface references to global interface references. A global interface references can only be created with some knowledge about the infrastructure. However, the application programmer does not have to worry about it as long as she uses interface references provided by the capsule (or a naming service) in a distributed context. The scope of a local interface reference is its local address space (or capsule). The scope of a global interface reference is global.

The *capsule proxy* is a local representation of a remote capsule. A capsule proxy can be created with some knowledge about the remote capsule. This knowledge includes the location (the node) of the remote capsule and where (the port) the capsule is listening for requests. A capsule is only available through a capsule proxy when it explicitly has made its services available by starting its serve loop. The services of the local capsule is always available for its components.

The differences between the access to the local capsule and to remote capsules (through a capsule proxy) are usually not visible for the application programmer. The interfaces of the local capsule and a capsule proxy are equal and often used transparently. Usually, the application programmer does not have to care if she is using the local capsule or a capsule proxy. This is another example of the access transparency provided by OOPP.

A capsule provides services to manage local bindings. These services are useful when a local binding in a remote capsule has to be established or broken. The request is then done through a capsule proxy.

A capsule also provides some low-level features used to call methods of the interfaces of registered components. These low-level-method-call features are mainly available for the programmer who wants to implement new types of bindings. They are not meant to be used by the application programmer. Three different types of invocations are available:

i) Synchronous messages where a call is done and the caller waits for a reply

ii) Asynchronous messages where the caller picks up the reply later

iii) Announcement where no reply is given (and no return values are possible)

The perceptive reader will see the similarity with these types of low-level invocations and the different types of operational bindings presented earlier.

4.3.2 Name server

How should a program get access to or locate a remote interface? A naming service provides the solution. It provides a way of getting access to remote interfaces based on an address or a name. The program applies a unique name or address identifying the interface and the naming service returns a (global) reference to this interface.

OOPP provides a simple naming service by one or more *name servers*. They are used to get access by names to interfaces and capsules in the form of interface references and capsule proxies respectively. A name server is identified by a node identifier (IP address) and a port number. A default port number is used if one is not specified (the default OOPP naming service port). A name server is accessed through a name server proxy. A name server proxy can be created from the name server proxy class when the node identifier and the (optional) port number are known. A different approach could be to identify a name server or even a service (interface) by an URL. This requires some additional support from the web servers.[2]

A capsule is *exported* (registered) in a OOPP name server with a name. An *import* method (a request) to the name server using this name will return a capsule proxy representing the registered capsule. The returned capsule proxy has an implicit binding to the capsule. The capsule proxy can be used immediately to access the services of its capsule. Similar export and import methods are available from the name server to every kind of interface in OOPP. The import method for interfaces returns a global interface reference. An implicit operational binding is automatically created to the remote interface. The returned interface reference can be used immediately to invoke the methods of the remote interface. This is similar to a proxy in CORBA terminology. The returned interface reference or proxy is a local representation of a remote interface or service. The remoteness of the interface or service is hidden by the proxy providing access transparency.

Sometimes the application programmer needs to control the creation of a binding between two (remote) interfaces. The name server supports this by providing a *lookup* method on interfaces. The lookup method returns an interface reference but no implicit binding is created. The returned interface reference can be used later to actually create a binding to the remote interface.

An important difference between the import and the lookup methods of a name server is the returned interface reference. The interface reference returned from a lookup method is a global interface reference for the remote interface exported. The interface reference returned from an import method is a matching (opposite) interface reference. For example, if the remote interface exports the methods f and g and imports no methods, the returned interface reference from a lookup also represents an interface that exports methods f and g and imports no methods. However, the interface reference returned from an import method represents an interface that exports no methods and imports method f and g.

4.3.3 Node manager

In the RM-ODP architecture document [62] a *node* is defined as a "configuration of engineering objects forming a single unit for the purpose of location in space, and which embodies a set of processing, storage and communication functions." An example of a node is a computer and its software. OOPP has adapted this definition of a node.

Each node in OOPP can serve several capsules. The *node managers* task is to ensure that the capsules can coexist on a node without unintentionally interfering with each other. This involves managing shared resources like connection ports on the node. The node managers are meant to be invisible and they can be completely ignored by the application programmers. Their services are automatically accessed by the infrastructure (capsules and low-level bindings). A node manager is also accessed through a proxy. Each capsule has a node manager proxy used to access its local node manager. The node manager is usually started automatically when the first capsule is started on a node.

4.3.4 Factories

New components and objects are usually introduced in an object oriented programming model with a class. In a programming model with distributed objects and composite components and possibilities for resource reservations and management this approach is not feasible. A new service feasible to perform these complex tasks is needed. This service is called a factory.

The task of a *factory* in OOPP is to add new components in capsules. Factories range from generic to specialised. A specialised factory create one type of components where the user specifies little or nothing to influence the creation process. A generic factory needs more information from the user and can create a much broader range of different components. A typical factory in OOPP creates a distributed composite component.

A *binding factory* is used to create explicit bindings. This usually involves deployment of components in different capsules, creating local bindings between interfaces in different capsules and creating low-level bindings between interfaces in different capsules. A factory often uses other factories to create some of the contained components of the resulting component.

Some standard factories are a part of OOPP. A *component factory* is provided to create primitive components. A *composite components factory* is provided to create generic composite components. In this case, the user have to specify the resulting composite component completely with a component graph description, its interfaces and the other factories (or classes) involved. Factories for the standard operational, signal and stream bindings are also provided.

4.4 Meta-models

OOPP provides two of the meta-models discussed earlier: the encapsulation and the composition meta-model. Each meta-model is accessed through *meta-objects*. A meta-object can also have a meta-level. Theoretically this recursion of accessing meta-levels of a meta-level can go on for ever. So rather than fixing the number of possible meta-levels in OOPP, meta-objects are created on-demand when they are accessed. This allows for an infinite number of meta-objects. In practice, this is of course never done. But the possibility of accessing the meta-levels of a meta-object is an feature of OOPP. It makes it possible to inspect and change the behaviour of a meta-object.

4.4.1 Encapsulation meta-model

The *encapsulation meta-model* provides access to the representation or the implementation of components, interfaces and objects. The encapsulation meta-model of a given interface, component or object is accessed through its encapsulation meta-object.

Components, interfaces and objects provide different meta-objects representing their encapsulation meta-models. The meta-models of objects are accessed using the standard encapsulation meta-object for objects. The meta-objects of components and interfaces provides more specialised interfaces based on their typical behaviour and characteristics. It is also possible to use the standard encapsulation meta-object for objects to perform a similar set operations on components and interfaces.

Encapsulation meta-model services for object

The encapsulation meta-objects for *objects* can be used

i) to inspect a given object (return a description of it),

ii) to manipulate its attributes,

iii) to manipulate its methods, and

iv) to completely change its implementation (class).

Inspection returns a description of the object including its class, attributes and methods. This is useful both for the inspection of the implementation (methods provided, attributes available and so on) and the current state of the object.

Methods can be added, overwritten, or deleted from the object. This is useful when a smaller modification of a given object is necessary or useful.

To a given method of an object pre-, post- and wrap-methods can be added. A pre-method of a method f is a method that is called before f is invoked. A pre-method of f can access and change the arguments of f. A post-method of f is a method that is called after f returns but before the result is returned to the caller. A post-method of f can access the arguments and the return value of f. It can even change the return value. A wrap-method of f wraps the invocation of f completely. A wrap-method of f has access to and can change the arguments and the return value of f. A wrap-method of f can even ignore to invoke f completely. Pre-, post- and wrap-methods have many potential usages. One is to monitor method calls and their arguments and return values and possibly change the arguments and return values in given circumstances. Another one is to guard and synchronise method calls.

An implementation of a new method has to be supplied when a method is added or overwritten in an object. This is also true for pre-, post- or wrap-methods when they are added to methods of the object.

It is also possible to add methods that are called every time a given attribute of an object is accessed or changed. Examples of the usage of these methods include the possibility to change the value returned when an attribute is accessed (read) and to monitor the value of an attribute when it is changed. Such a monitor could raise an alarm when a given threshold value is reached. The programmer should be aware of the extra cost of accessing and changing attributes of an object when such attribute methods are added.

Encapsulation meta-model services for interfaces

The encapsulation meta-object for *interfaces* can be used to

i) inspect a given interface (return a description of it),

ii) manipulate its exported and imported methods, and

iii) change the object the interface belongs to (are connected to).

Inspection returns a description of the interface including its exported methods, imported methods, and the object it belongs to. The encapsulation meta-objects of interfaces distinguish between exported and imported methods. They provide services to add or delete exported and imported methods in an interface.

Since an interface is a mapping to actual methods implemented in objects, an implementation of methods are not supplied when methods are added to the exported or imported methods of an interface. It is also possible to add pre-, post- and wrap-methods to methods in an interface. In these circumstances an implementation of pre-, post- and wrap-methods has to be supplied. Their implementation is similar to the pre-, post- and wrap-methods of objects.

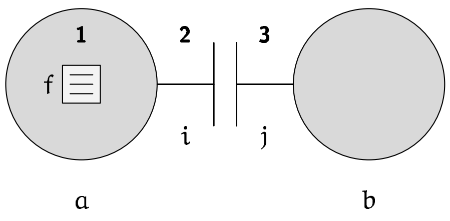


Figure 4.5 Different locations to add pre-, post- and wrap-methods.

Encapsulation meta-model services for components

The encapsulation meta-object of *components* is used to inspect, add, remove and replace (external) interfaces of a component. The composition meta-model (see below) provides the services needed to manipulate the component graph of a composite component.

Examples of the usage of the encapsulation meta-model

Figure 4.5 illustrates all the different locations where pre-, post-, and wrap-methods can be added. The example contains an object a with a method f. Interface i exports method f and interface j imports method f. Interface i and j are connected with a local binding. The encapsulation meta-object e_a for a can be used to add pre-, post- and wrap-methods to f in object a (**1** in Figure 4.5). The encapsulation meta-object e_i for i can be used to add pre-, post- and wrap-methods to the exported method f in interface i (**2** in Figure 4.5). Finally, the encapsulation meta-object e_j for j can be used to add pre-methods, post-methods and wrap-methods to the imported method f in interface j (**3** in Figure 4.5).

Given two interfaces i and j are bound with a local binding where interface i exports and interface j imports the method f and g. Suppose a need to extend the interfaces with a method h emerges. The encapsulation meta-object e_i of interface i and the encapsulation meta-object e_j of interface j are fetched (created on demand). e_i is then used to add method h as an exported method of interface i. e_j is used to add method h as an imported method of interface j. Interface j can now be used to call the method h exported from interface i. Notice that there were no need to break and then later rebind the local binding between interface i and j. Also notice that the object connected to interface i has to implement a method h (or this method has to be added using the encapsulation meta-object of the object).

Suppose object s implements a stack with the size n. The attribute c of object s is the number of elements currently on the stack. A system needs to monitor the stack and send a signal every time the stack is more than 90% full. The encapsulation meta-object e_s of the stack s is fetched. e_s is then used to add a method m to be called each time attribute c of s is given a new value. Method m checks if $c \geq 0.9n$ and sends a signal if this is true.

Finally, suppose a profiling task needs to monitor every method call to an object o. The encapsulation meta-object e_o is fetched and e_o is used to get a list of every method in object o. This list and the encapsulation meta-object e_o is then used to add a post-method p to every method in o. p forwards information about every method call to the profiling object. The information forwarded typically includes the name of the method called, the time of the call, all the arguments of the call, and the (optional) return value of the call.

4.4.2 Composition meta-model

Composite components typically represents complex binding objects. The adaptability needed for multimedia applications in a mobile environment identifies a need to manipulate and restructure (change and extend) the component graph of composite components during their life cycle. A *composition meta-model* is provided to achieve this.

The composition meta-model is empty for all non-composite components. It provides access to the component graph representing the composite component. Every external interface of the composite component and the composite component itself return the same composition meta-object. The components found in the component graph of one composite component are not necessary located in the same capsule.

The services provided by the composition meta-model are strongly influenced by the services presented in the Adapt project[3] for the manipulation of object graphs of open bindings [51].

Composition meta-object services

The composition meta-object can be used to

i) inspect the component graph,

ii) add and remove local bindings from the component graph, and

iii) insert, remove and replace components in the component graph.

Inspection returns a view of the component graph containing all components and all bindings of the composite component. This view can be used to get access to specific components or bindings, or to traverse the complete graph itself.

The composition meta-object can be used to create a new local binding between two (internal) interfaces in the composite component. This new local binding is added to the component graph. The composition meta-object can also be used to break existing local bindings between two (internal) interfaces in the composite component. This local binding is removed from the component graph.

The composite meta-object can be used to insert, remove and replace components contained in a composite component. A new component inserted in the composite component is added to the component graph. The new component is bound to existing interfaces in the component graph as specified in the call. Existing bindings will be removed if any of the existing interfaces already are bound to other interfaces. This service can for example be used to insert a filter component in a flow. A component removed from the composite component is completely removed from the component graph. Any bindings between interfaces of the removed component and interfaces of other components in the graph will be removed too. Replace is a mixture of the two operations above. It removes an existing component in the component graph and replaces it with a new one. Existing bindings to the original component are removed and new bindings are created to the new one. The new and the original component must be similar (including equal interfaces). The new component inherits the role of the original component.

Examples of the usage of the composite meta-model

A news broadcast with video and audio is transmitted as an MPEG stream [37] from a server to a laptop computer. When the laptop is disconnected from the high-bandwidth local area network, a low-bandwidth wireless network automatically replaces it. The new low-bandwidth connection is not feasible for the transmission of the MPEG video stream, and the news broadcast client on the laptop that has observed the problems introduced by the network replacement wants to filter out the video and only receive the audio. The news broadcast server and client are bound with an MPEG stream object. This stream object is a composite component containing a MPEG encoder, a sender object (on the server side), a receiver object and an MPEG decoder (on the client side). The sender and receiver objects are bound with an UDP/IP binding. The client uses the composition function to fetch the composite meta-object of the stream binding. It then uses the insert method of the meta-object to insert a filter object between the MPEG encoder object and the sender object (on the server side). This filter object removes the video from the MPEG stream, and only the (low data volume) audio will be transmitted to the client.

Another client of the news broadcast described above will first try lower the need of bandwidth with a lower quality video. This client uses the composition function to fetch the composite meta-object of the stream binding. It then uses the replace method of the meta-object to replace the MPEG encoder with a H.263 encoder and the MPEG decoder with a H.263 decoder [64].

Notes

1. The term "matching interfaces" used in this chapter is rather vague. A more precise definition follows. Two interfaces i and j where interface i exports the set of methods E_i and imports the set of methods M_i and interface j exports the set of methods E_j and imports the set of methods M_j. Interface i and j match (denoted $i \bowtie j$) if the following two conditions holds:

$$\forall m_i \in M_i, \exists e_j \in E_j, m_i \subseteq e_j \tag{4.1}$$

$$\forall m_j \in M_j, \exists e_i \in E_i, m_j \subseteq e_i \tag{4.2}$$

The meaning of an expression like $f \subseteq g$ is that "the signature of (exported) method g fulfils the signature of (imported) method f". A simple approach only checks if the method names of g and f are equal. A more refined approach could check if the type of the arguments of method g and f match. This could be further extended with sub-typing or compatibility rules [19, 48]. The expression $i \lhd j$ where i and j are interfaces should be interpreted as equation (4.1) and the expression $i \rhd j$ should be interpreted as equation (4.2). This is a weaker demand on the relations between interfaces than $i \bowtie j$ and it is useful to check interfaces in one-way bindings (a binding where the only interest is in the exported methods of one interface and the imported methods of the other interface).

2. One such an approach is the Z Object Publishing Environment (Zope, see http://www.zope.org/). Zope is an object-based web application platform that provides direct URL access to objects. The URL '/obj/meth?arg=val' calls the method 'meth' of the object 'obj', and passes the argument 'arg' with the value 'val' to the method.

3. Adapt is a research project at the distributed multimedia research group in Lancaster. The project is closely related to the Open-ORB project. The main aim of the Adapt project is to investigate the development of distributed systems support to manage the differing levels of connectivity a mobile user will experience. More specifically, Adapt is investigating the required support for adaptive multimedia applications which are capable of intelligently adapting to quality of service fluctuations. See http://www.comp.lancs.ac.uk/computing/research/mpg/index_mods.html for more details.

Chapter 5

OOPP implementation

A description of a prototype implementation of the Open-ORB architecture described in chapter 3 now follows. The design issues of this prototype have been discussed in chapter 4. The motivations for this prototype is to validate the Open-ORB architecture and to illustrate the utility of various adaptation mechanisms. The prototype implementation should also provide an expressive programming model that supports the development of distributed applications. This prototype will later be used to investigate the possibilities to introduce an automata based quality of service management scheme (see chapter 6).

This reflective middleware platform prototype has been implemented in Python[1] [106, 76] on top of the socket interface of the TCP/IP and UDP/IP protocols. Python is an object-oriented scripting language well suited for prototyping. The possibility to use the interpreter interactively and test segments of the code individually is a great advantage during the development of complex systems like OOPP. The language provides some access to the implementation of its language constructs, like objects and classes. This openness makes it easier to add reflective features needed in the OOPP programming model. Python also provides a short development cycle from ideas to running code. This was also considered a great advantage in the development of the prototype.

The Open-ORB prototype described here implements an RM-ODP [50, 60, 93] inspired programming model with programming structures and their infrastructure. Reflection is provided through the implementation of two distinct meta-models: the encapsulation and the composition meta-model. The other two meta-models discussed in chapter 3 are not implemented in this prototype.

Table 5.1 summaries the argument and return value types used in the following description (signatures) of the implementation. The OOPP implementation is split into several modules. These modules are organised in three packages. The core package provides the core programming model including the programming structures and their infrastructure. The meta package provides the meta-programming model (the reflective features). Finally, the mngt package provides the QoS management part (see Chapter 6).

i	Interface references	n	Name server proxy
x	Explained in the text	v	An attribute value
l	Local binding control object	r	Result value
o	Object instance	k	Key or name
c	Component instances	m	Method name
b	Binding object	f	Method implementation
u	Unique component identifier	a	Argument tuple
C	Class or factory	w	Argument dictionary
p	Capsule proxy	I	Inspect dictionary
e	Encapsulation meta-objects	g	Composition meta-objects
$[x]$	A list of x	x^*	x is optional
$\{k : x\}$	Map (dictionary) from k to x	(x, y)	A two-tuple with x and y

Table 5.1 List of arguments and return values used in descriptions (signatures) in the following text. Suffixes (like x_1 and x_2) are used to distinguish between different values of equal type.

5.1 The programming structures

An application programmer uses the *programming structures* to build applications. As described in the previous chapter, OOPP provides components, interfaces and bindings as its basic programming structures. It also inherits the object and class terminology from Python.

5.1.1 Interfaces and local bindings

An *interface* provides a set of exported and imported methods and it is connected (related) to a given object. This is illustrated in the simple example from Figure 4.1 in the previous chapter. An object a implements a method f and have an interface i. Method f is available for other objects through interface i if interface i exports it. An object b has an interface j that imports a method f. Object b uses interface j to call an external method f implemented in an object with an interface that exports a method f. This other object could be object a with interface i. Interface i and j are created with the basic interface reference class IRef. Line 1 and 2 in Listing 5.1 (a) are the Python code used to create them. The constructor of the IRef class will check that object a actually implements the exported method f when the interface reference i is created. An exception is raised if this is not the case.

It is important to observe that when two objects with matching interfaces as described above have been created, there is no association between them until they are (explicitly) bound to each other. The localBind function creates a *local binding* between two interfaces. Line 3 in Listing 5.1 (a) creates a local binding between interface i and j. The localBind function returns a *local binding control object*. This control object only has a limited functionality in the current implementation. A local binding only exists as cross references

```
i = IRef(a, ["f"], [])          1
j = IRef(b, [], ["f"])          2
lb = localBind(i, j)            3
result = j.f()                  4
```

(a) Create a local binding

```
breakBinding(i, j)              1

lb.breakBinding()               1
```

(b) Break a local binding

Listing 5.1 Create interfaces and a local binding and later break the local bindings.

between the two interfaces bound. The binding will not be broken if the local binding control object is deleted or garbage collected. A one-way local binding is created with the localBindOneWay function. It only connects the exported methods of argument one with the imported methods of argument two (and not vice versa).

A breakBinding function is used to break (disconnect) a local binding. This is either done directly on the interfaces using their interface references as arguments to the breakBinding function (line 1 of listing 5.1 (b)) or with the breakBinding method of the local binding control object (line 2 of listing 5.1 (b)). The programmer selects freely the most convenient method to her own liking.

You can also reuse a local binding control object. The reBind and reBindOneWay methods of the control object are used to create a new local binding in control of the existing local binding control object.

Interfaces are accessed through *interface references* like i and j from Listing 5.1. An interface reference also has a reference to an *interface object*. The creation of a local binding populates the name space of the interface reference with forwarding methods for its imported methods. These method calls are forwarded to the interface object of the other interface. The name space of an interface object contains forwarding methods for the exported methods of its interface. These forwarding methods forward the method calls to the actual methods of the object. These forwarding methods are populated in the interface object when the interface reference is created. An exception is raised if the actual method does not exists in the object.

Figure 5.1 illustrates this for the local binding example used above. The interface of object a is represented with the interface reference i and its interface object o_i. The interface object o_i contains a forwarding method f_{o_i} to method f of object a. The interface of object b is represented with the interface reference j and its interface object o_j. Interface reference j contains a forwarding method f_j to the forwarding method f_{o_i} in interface object o_i when a local binding between i and j exists.

The perceptive reader has observed that the forwarding method f_{o_i} in interface object o_i is not necessarily needed in the implementation of a local binding. Since the interface reference j and method f in object a are located in the same address space the forwarding method f_j in interface reference j could alternatively have a direct reference to method f

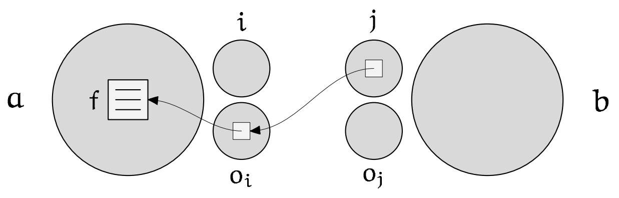


Figure 5.1 The implementation of a local binding between interface i and j where o_i and o_j are the interface objects of interface i and j respectively.

```
src = StreamSrcIRef(a)
sink = StreamSinkIRef(b)
lb = localBind(src, sink)
src.put(data)
```

(a) Stream interfaces

```
src = SigSrcIRef(a)
sink = SigSinkIRef(b)
lb = localBind(src, sink)
src.event()
```

(b) Signal interfaces

Listing 5.2 The creation of stream and signal interfaces.

in object a. This approach would have reduced the level of indirection by one. However, the possibility to have fine grained control over everything that happens on both sides of the interface would be lost. A method call to method f in object a would not pass through interface i (or more precisely, interface object o_i) at all.

As mentioned earlier, an interface can both export and import methods. It is, however, possible for a programmer to choose a programming style where interfaces always either export or import methods. Stream and signal interfaces follow this style. Two classes CIRef and SIRef are provided to make it more convenient to choose this programming style. These classes are a specialisation of the standard IRef class described above. They represent the client and the server side in a traditional client-server setup. An interface reference created with the CIRef class is similar to an interface reference created with the IRef class when no exported methods are specified. An interface reference created with the SIRef class is similar to an interface reference created with the IRef class when no imported methods are specified.

The bindIRef function is just a combination of creating an interface reference and connect it (with a local binding) to another interface. The first argument is the interface reference of the interface that the new interface should be connected to. The second optional argument is the object the new interface should be related with (possible none). Line 2 and 3 of listing 5.1 (a) can be replaced by the statement 'j = bindIRef(i, b)' (the reference to the local binding control object lb would be lost).

Stream and signal interfaces exist in a source and a sink pair. They are a specialisation of the operational interfaces presented above. A local binding between a source and a sink

$\mathsf{IRef}(o, [m]_1, [m]_2) \rightarrow i$	Constructor of the IRef class. o is the object i is connected to (possibly empty when i only imports methods). $[m]_1$ lists all the exported methods of i. $[m]_2$ lists all the imported methods of i.
$\mathsf{CIRef}(o, [m]) \rightarrow i$	Constructor of the CIRef class. o is the object i is connected to. $[m]$ lists all the imported methods of i.
$\mathsf{SIRef}(o, [m]) \rightarrow i$	Constructor of the SIRef class. o is the object i is connected to. $[m]$ lists all the exported methods of i.
$\mathsf{LBindCtrl}(p, i_1, i_2) \rightarrow l$	Constructor of the LBindCtrl class. p is a reference to the capsule of the binding (only used when the local binding is actually created by a capsule). i_1 and i_2 are the interface references of the two interfaces to be bound.
$l.\mathsf{breakBinding}()$	Break local binding controlled by l.
$l.\mathsf{reBind}(i_1, i_2)$	Create a local binding between i_1 and i_2 controlled by l.
$l.\mathsf{reBindOneWay}(i_1, i_2)$	Create a one-way local binding between i_1 and i_2.
$\mathsf{localBind}(i_1, i_2) \rightarrow l$	Create a local binding between i_1 and i_2.
$\mathsf{localBindOneWay}(i_1, i_2) \rightarrow l$	Create a one-way local binding between i_1 and i_2.
$\mathsf{breakBinding}(i_1, i_2)$	Break local binding between i_1 and i_2.
$\mathsf{bindIRef}(i_1, o^*) \rightarrow i_2$	This function creates interface i_2 as a matching interface for i_1. It then connects the two interfaces (with a local binding). The optional argument o is the object the new interface should be related with.

Table 5.2 Services available from the lbind module.

interface pair is also created with the localBind function. Listing 5.2 (a) contains the code that creates a stream interface reference source and sink pair and makes a local binding between them. The source interface belongs to object a and the sink interface belongs to object b. A stream interface uses a put method with one data argument. Object b then has to implement such a put method. A signal interface uses a method event without any arguments to send a signal. One signal interface only supports one type of signals. Listing 5.2 (b) contains the code that creates a signal interface reference source and sink pair and makes a local binding between them. In this example object b has to implement an event method (with no arguments).

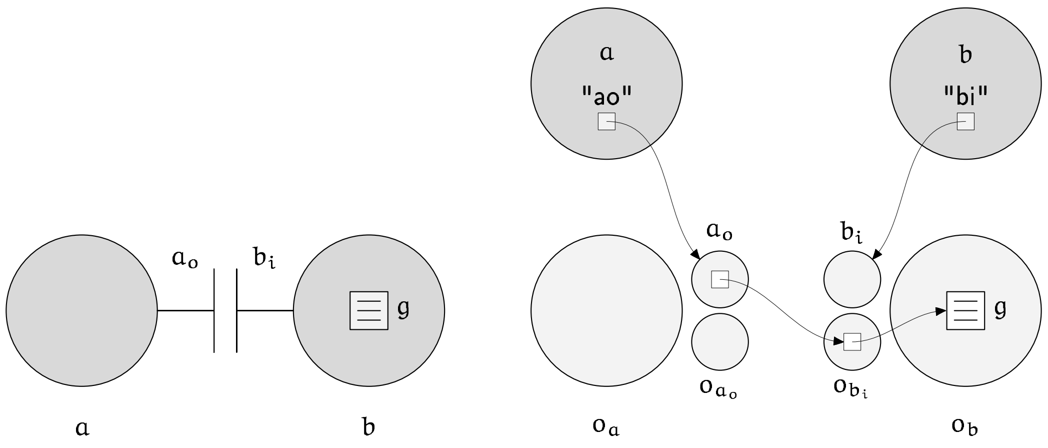
The standard interface references and the local binding function are implemented in the lbind module in the core package of OOPP. The services provided by the lbind module are listed in Table 5.2.

5.1.2 Components and composite components

Components are the main building blocks in OOPP. Listing 5.3 contains code that creates two components a and b using the Component class. The first argument of the Component constructor (inside $\{\,\}$) contains the mapping from the interface of the object (i and j) to the interface of the component (a_o and b_i). The second argument is the contained object (o_a and o_b). A local binding between interface a_o of component a and interface b_i of

```
a = Component({"ao": i}, oa)
b = Component({"bi": j}, ob)
lb = localBind(a.interfaces["ao"], b.interfaces["bi"])
```

Listing 5.3 Create component a and b and a local binding.



(a) Programmer's view **(b)** Implementation details

Figure 5.2 Two components a and b connected with a local binding.

component b is also created. Figure 5.2 (a) illustrates this setup. Component a and b are created with the existing objects o_a and o_b. These objects implements the methods exported by the interfaces of the components and uses the methods imported by the interfaces of the components. Component a and b wrap object o_a and o_b respectively. Figure 5.2 (b) shows the implementation details of these components and local binding between their interfaces.

A *composite component* contains a set of other components represented with a component graph. Composite components can represent a complex structure of components as a single component with a set of well defined external interfaces.

The *component graph* specifies the *contained components* and the local bindings between the different internal interfaces of the contained components in the composite component. A component contained in a composite component can again be a composite component. This recursive component containment is terminated by *primitive components* (a component that is not a composite component). The set of interfaces provided by a composite component are a mapping from a subset of the interfaces of its contained components. This implies that an interface provided by a composite component is a mapping to an interface in one of the contained components. Figure 5.3 shows a simple composite component c with two (external) interfaces c_i and c_o, two contained components a and b and only one

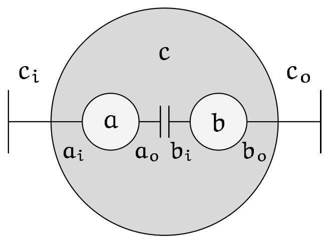


Figure 5.3 A composite component c containing two components a and b.

```
c = Composite(
    {"ci": ("a", "ai"), "co": ("b", "bo")},
    {"comps": {"a": a, "b": b},
     "iif": {"ao": ("a", "ao"), "bi": ("b", "bi")},
     "edges": [("ao", "bi")]})
```

Listing 5.4 Create the composite component from Figure 5.3.

local binding between interface a_o and b_i in the object graph. The external interface c_i is a mapping to interface a_i of component a and the external interface c_o is a mapping to interface b_o of component b. The generic Composite class can be used to create any composite component. Listing 5.4 creates composite component c from Figure 5.3. The first argument (inside { }) specifies the external interfaces c_i and c_o. The second (and last) argument specifies the contained components and their component graph. "comps" lists and gives an internal name to all the contained components, "iif" list all internal interfaces (given a name) and "edges" specifies the local bindings in the component graph using the names from "iff". "iff" in this case indicates that the name "ao" will be used for interface a_o of component a and the name "bi" will be used for interface b_i of component b.

Figure 5.4 exposes some of the implementation details of composite component c. The contained components a and b of composite component c contains object o_a and o_b providing the implementation (code) of these components. Interface c_i of component c is a mapping to interface a_i of component a. Interface a_i of component a is a mapping to interface a_i of object o_a. We have a similar mapping of the other external and internal interfaces of composite component c.

The object graph is not directly exposed in Figure 5.4. It will contain the components a and b, the list of internal interfaces a_o and b_i, and a mapping from the name of the local bindings (edges) to the control object of each such binding.

A composite component can be distributed. A distributed composite component contains components located in different address spaces (capsules). These address spaces can again be located on different nodes. Components in different address spaces can be bound with (non-local) bindings.

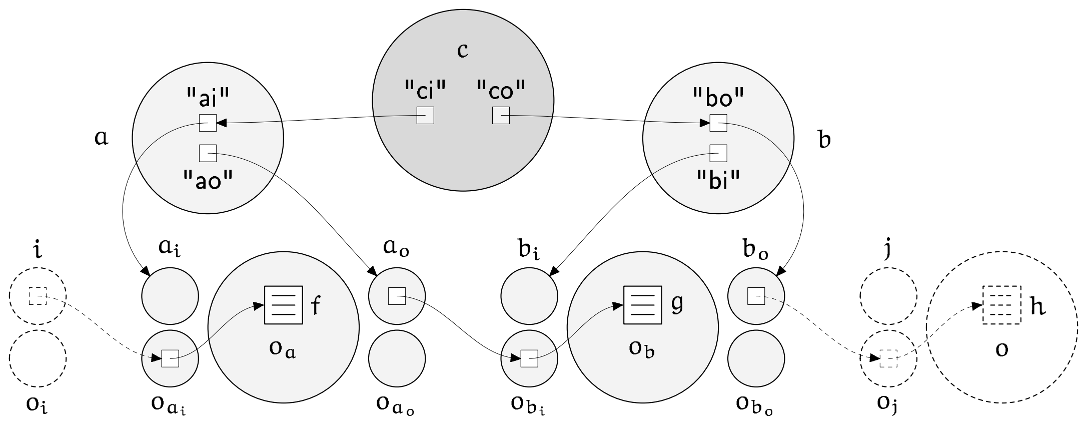


Figure 5.4 The implementation details of the composite component from Figure 5.3.

Table 5.3 summaries the services available from the component and composite module in the core package of OOPP.

5.1.3 Bindings

Bindings provide connections (communication) between objects in different address spaces and on different nodes. A typical binding is a distributed composite component containing stubs and a lower level binding. Figure 4.3 in the previous chapter showed an example of a binding object containing two stubs and a TCP/IP binding. OOPP provides three different kinds of bindings.

An *operational binding* can be created with the constructor of the OpBinding class. However, it is expected that the application programmer will use the remoteBind factory since its usage is simpler than the class constructor and similar to the localBind function. The remoteBind function creates an operational binding using the OpBinding class. It then uses this operational binding to connect (bind) the two given interfaces.

Listing 5.5 contains the code used to create an operational binding between interface i and j and then call method f exported by interface i. Interface i and j belongs to two different components located in different address spaces. Interface i is the remote interface and interface j is the local interface in the code in Listing 5.5. The two interface references used as arguments to the constructor of the OpBinding class are only used to provide the location and signatures of the interfaces. This information is used to create stubs in the right capsules and with the right signatures. The localBind functions has to be used later to actually connect the two interfaces using the operational binding (see line 2 and 3 of Listing 5.5 (a)). As mentioned above, the remoteBind factory combines these operations in a single statement (see Listing 5.5 (b)). Notice that the remoteBind factory does not return a component instance of the OpBinding class. The returned value is a unique identifier for this component (see details in section 5.2.1 below).

Component($\{k : i\}, o) \rightarrow c$	Constructor of the Component class. The dictionary $\{k : i\}$ lists all the interfaces of c. Each k is a internal name of an interface represented by the interface reference i. o is the object implementing the interfaces (or at least the exported methods of the interfaces).
componentFactory($[k], C, a^*, w^*) \rightarrow c$	The generic component factory. The list $[k]$ lists all the interfaces by their (internal) name. C is the class (or factory) of the object implementing these interfaces and a and w are the possible arguments to the constructor/factory C.
Composite($\{k_1 : (k_2, k_3)\}, \{k_4 : x\}) \rightarrow c$	Constructor of the Composite class. The dictionary $\{k_1 : (k_2, k_3)\}$ lists all the external interfaces of c. Each external interface k_1 is a mapping to an interface k_3 of a contained component k_2. In the $\{k_4 : x\}$ argument the following keys (k_4) with their related values should be provided: "comps", "iif" and "edges".
compositeFactory($\{k_1 : (k_2, k_3)\}, \{k_4 : x\}) \rightarrow c$	The generic composite component factory. The description of the arguments matches the description of the arguments of the constructor of the Composite class above.
c.interfaces	A mapping to the interfaces of component c.

Table 5.3 Services available from the component and composite modules.

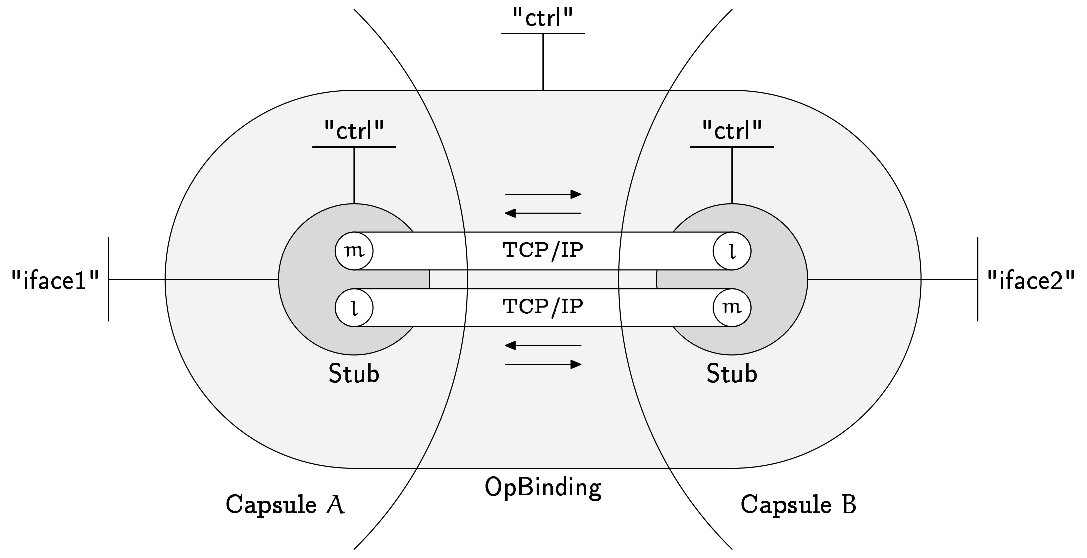


Figure 5.5 Implementation of binding object.

```
b = OpBinding(i, j)
la = localBind(i, b.interfaces["iface1"])
lb = localBind(b.interfaces["iface2"], j)
result = j.f()
```

(a) Using OpBinding class

```
u = remoteBind(i, j)
result = j.f()
```

(b) Using remoteBind factory

Listing 5.5 Creating and using an operational binding.

Figure 5.5 illustrates how the operational binding is implemented. Capsule A and capsule B are two different address spaces. The operational binding is a composite component containing two stub components (or just stubs for short). It provides three external interfaces, one control interface "ctrl" and two connecting interfaces "iface1" and "iface2" (the names of these interfaces are given by the OpBinding class). The two stubs are connected with two low level TCP/IP connections. One is for requests (method calls) from an object in capsule A to an object in capsule B (with reply back to the object in capsule A) and the other one is for requests from an object in capsule B to an object in capsule A (also with reply). Similar to local bindings, the operational bindings are two-way bindings (method calls can go in both directions). The m objects are message objects (send request and receive reply) and the l objects are listen objects (receive request and send reply). A message object is connected with a low level TCP/IP connection to a listen object in the remote address space.

Two interfaces references i_1 and i_2 are used as arguments to the constructor of the OpBinding class. These interface references contain the following information used by the constructor: (i) the location of interface i_1, (ii) the location of interface i_2, (iii) the exported and imported methods of interface i_1, and (iv) the exported and imported methods of interface i_2. The constructor uses this information to know where the stubs of the binding should be created and what method calls it should provide (forward) in what direction. It uses the encapsulation meta-model (see Section 5.3.1 below) of the stubs and its interfaces to install forwarding methods for the exported and imported methods of the interfaces i_1 and i_2. It uses the services of the possibly remote capsules (see Section 5.2.1 below) to create instances of the stubs at their given location.

A *signal binding* can be created with the constructor of the SigBinding class and a *stream binding* can be created with the constructor of the StreamBinding class. However, the corresponding factories sigBind and streamBind are easier to use and are expected to be preferred by the application programmers.

Listing 5.6 contains the code used to create and use signal and stream bindings. The arguments to the SigBinding and StreamBinding constructors are capsule proxies (or the local capsule) used to locate the source and sink side of the bindings. The localBind function is then later used to connect the source and sink interfaces through the binding (see Listing 5.6 (a) and Listing 5.6 (c)). The sigBind and streamBind factories combine

```
b = SigBinding(pa, pb)
la = localBind(src, b.interfaces["src"])
lb = localBind(b.interfaces["sink"], sink)
src.event()
```

(a) Using SigBinding class

```
u = sigBind(src, sink)
src.event()
```

(b) Using signal factory

```
b = StreamBinding(pa, pb)
la = localBind(src, b.interfaces["src"])
lb = localBind(b.interfaces["sink"], sink)
while 1:
    src.put(getdata())
```

(c) Using StreamBinding class

```
u = streamBind(src, sink)
while 1:
    src.put(getdata())
```

(d) Using stream factory

Listing 5.6 Creating and using signal and stream bindings.

OpBinding(i_1, i_2) $\rightarrow$ b	Constructor of the OpBinding class. i_1 and i_2 are the interface references of the two interfaces to be bound. This only creates the binding. Interface i_1 and i_2 are not bound.
remoteBind(i_1, i_2) $\rightarrow$ u	Create a remote binding between interface i_1 and i_2.
SigBinding(i_1, i_2) $\rightarrow$ b	Constructor of the SigBinding class. i_1 and i_2 are the interface references of the two interfaces to be bound.
sigBind(i_1, i_2) $\rightarrow$ u	Create a signal binding between interface i_1 and i_2.
StreamBinding(i_1, i_2) $\rightarrow$ b	Constructor of the StreamBinding class. i_1 and i_2 are the interface references of the two interfaces to be bound.
streamBind(i_1, i_2) $\rightarrow$ u	Create a stream binding between interface i_1 and i_2.
b.serve()	Make the binding ready for request.
b.stopserve()	Make the binding unavailable.
b.start()	Start the flow (stream binding).
b.stop()	Stop the flow (stream binding).
b.interfaces	Contains the interfaces of the binding b. An instance of the OpBinding class contains the interfaces "iface1", "iface2" and "ctrl". An instance of the SigBinding or StreamBinding class contains the interfaces "src", "sink" and "ctrl".

Table 5.4 Services available from the different binding modules (opbind, sigbind and streambind modules). The serve(), stopserve(), start() and stop() methods are available from the "ctrl" interface of the bindings.

Capsule() $\rightarrow p$	Constructor of the Capsule class. Never used by the application programmer (automatically called when the capsule module is imported).
CapsuleProxy(k, v) $\rightarrow p$	Constructor of the CapsuleProxy class. The first argument (k) is its node (host-name or IP-address) and the second argument (v) is its listening port number.
equalCapsule(i_1, i_2) $\rightarrow v$	Check if interface i_1 and i_2 are located in the same capsule.
local	Object exposing the services of the local capsule (an instance of the Capsule class).

Table 5.5 Services available from the capsule module. See Table 5.6 for the services available from a capsule and a capsule proxy.

these operations in a single statement (see Listing 5.6 (b) and Listing 5.6 (d)).

The implementation details of the signal and stream bindings are similar but simpler compared to the generic operational binding presented above. The signal binding only has one pair of the message and listen objects (m and l in Figure 5.5) and only one TCP/IP connection. The main interfaces of the binding are called "src" and "sink" and not "iface1" and "iface2". The stubs of a signal binding are a source stub on the source side and a sink stub on the sink side. The stubs of a stream binding are also a source and a sink pair. The source side contains a flow source connected to the sink side flow sink with a UDP/IP connection. This simple stream binding does not contain any buffers or knowledge about time (buffering and knowledge of time are usually important for satisfying transfer of continuous media over a stream binding).

Table 5.4 lists the services provided from the opbind, sigbind and streambind modules in the core package of OOPP.

5.2 The infrastructure

A component becomes a part of OOPP *infrastructure* when it is located in a capsule. The infrastructure provides different services for its components. These services include life-cycle functions, naming and low-level interaction.

5.2.1 Capsule

A *capsule* is a managed address space providing a set of services for local and remote components. Local components access the capsule services through the local object of the capsule module. Remote components access the capsule services through a *capsule proxy*.

A capsule proxy can only be used to access a remote capsule if the remote capsule has started its serving loop. This has to be done explicitly with the serve method locally. This means that any capsule is unavailable remotely until the serving loop has been started. The stopserve method is used to stop the serving loop.

p.serve()	Start serving requests from remote components
p.stopserve()	Stop serving requests from remote components
p.registerComponent(c) $\rightarrow u$	Register component c in capsule.
p.getIRef(u, k) $\rightarrow i$	Get interface reference with key k from component u.
p.mkComponent(C, a^*, w^*) $\rightarrow u$	Create and register a component with class or factory C using arguments a or w.
p.delComponent(u)	Delete registered component u.
p.rcpComponent(u, p_1)	Make a copy of component u in capsule p_1.
p.migrateComponet(u, p_1)	Move (migrate) component u to capsule p_1.
p.callMethod(u, k, m, a^*, w^*) $\rightarrow r^*$	Call remote method m in interface k of component u with arguments a or w.
p.announceMethod(u, k, m, a^*, w^*)	Remote announcement (method call without reply).
p.announceThread(u, k, m, a^*, w^*)	Remote announcement in separate thread.
p.sendMethod(u, k, m, a^*, w^*) $\rightarrow v$	Call remote method but do not wait for a reply.
p.recvMethod(v) $\rightarrow r$	Get reply from earlier sendMethod call. v is the handle returned from the previous sendMethod call.
p.localBind(i_1, i_2) $\rightarrow l$	Create a local binding between interface i_1 and i_2.
p.localBindOneWay(i_1, i_2) $\rightarrow l$	Create a one-way local binding between interface i_1 and i_2.
p.breakBinding(i_1, i_2)	Break a local binding between interface i_1 and i_2.

Table 5.6 Services provided from capsules and capsule proxies.

Table 5.5 summaries the services available from the capsule module in the core package OOPP. Table 5.6 lists services provided by capsules and capsule proxies. A component has to be registered in the capsule to be available for the services provided. A component c can be registered with the registerComponent method. A component created with the mkComponent method of the capsule will automatically be registered in the capsule. The mkComponent method has an argument C that is the class or factory used to actually create the component and two optional arguments a and w used to pass the arguments to the class constructor or the factory. Argument a is a tuple containing the arguments and argument w is a key-word list version of the arguments (one or none is used). The registerComponent and mkComponent methods return a local unique identifier u for the component (unique in this capsule). This identifier together with a capsule proxy provides a global identification of a component. The capsule also provides services to copy and migrate a registered component.

The getIRef method of a capsule returns a global interface reference i of the interface named k in the registered component u. Such interface references can be passed between different capsules (for instance as arguments in a remote method call). The capsule also provides services to establish and break local bindings in the capsule. These services are useful when a local binding in a remote capsule has to be established or broken. The request is then done through a capsule proxy. These methods have the same arguments as the functions with the same names discussed in section 5.1.1. The localBind and breakBinding functions presented in section 5.1.1 will use a capsule proxy to create a remote local binding when

```
import capsule                                1
u = capsule.local.registerComponent(c)        2
i = capsule.local.getIRef(u,"i")              3
p = capsule.CapsuleProxy("oo.no",1001)        4
```

(a) Use capsule services

```
u = p.mkComponent(C)                          1
r = p.callMethod(u,"i","f",("x",2))           2
b = remoteBind(p.getIRef(u,"i"),j)            3
r = j.f("x",2)                                4
```

(b) Call remote method f

Listing 5.7 Examples on the usage of capsule services.

they discover that the interfaces are located remotely.

Three different types of low-level invocations are available. A synchronous invocation sends a message and receives a reply with the callMethod method. A asynchronous invocation (also sometimes called a deferred synchronous invocation [49]) sends the message with the sendMethod method and later picks up the reply with the recvMethod method. Announcement can either be done with the announceMethod method or the announceThread method. The announceMethod method sends the message but does not wait for a reply. The announceThread method starts a new thread to send the message. These low-level messages are not meant for the application programmer. They are usually used by programmers that are creating new types of bindings (classes or factories) or other supporting functionality. The argument u in these message methods in Table 5.6 is the unique identifier of the component. The identifier is unique in the context of its local capsule (a global identification of the component is given combining this identifier and the capsule proxy representing the local capsule of the component). Argument k is the name (key) of the interface reference and argument m is the name of the method to be called. The last two arguments a and w are two optional arguments used to pass arguments to the actual method calls.

The capsule is created when the capsule module is loaded (imported first time). The local attribute of the capsule module represents the local capsule. Notice that the programmer should never use the constructor of the Capsule class. A capsule is automatically created the first time the capsule module is imported. The programmer only creates capsule proxies to get access to (remote) capsules.

Listing 5.7 (a) shows how the capsule is loaded (line 1), how component c is registered in the local capsule (line 2), how a global interface reference for interface i in component c is fetched (line 3), and how a capsule proxy p for a remote capsule at node "oo.no" is created (line 4). The last argument of the CapsuleProxy constructor is the port number the remote capsule is listening on. It will be shown later how to get access to a remote capsule without this knowledge (see name server in section 5.2.2 below).

Listing 5.7 (b) illustrates how the capsule proxy p can be used. In the first line the capsule proxy is used to create a component in the remote capsule. Method f exported in interface i of this component is then called with the low-level callMethod call (line 2).

NameServerProxy(k^*, v^*) → n	Constructor of the NameServerProxy class. k is the node (default is the local node) and v is the port number the name server is listening on (default a standard port number).
n.exportCaps(k, p)	Export capsule p with the key k.
n.exportIRef(k, i)	Export interface reference i with the key k.
n.delCaps(k)	Remove the exported capsule with the key k.
n.delIRef(k)	Remove the exported interface reference with the key k.
n.importCaps(k) → p	Return a capsule proxy with the key k.
n.lookupIRef(k) → i	Return the interface reference with the key k.
n.importIRef(k) → i	Return an interface reference with an implicit binding to the exported interface.
n.listCaps() → [p]	List all exported capsules in this name server.
n.listIRefs() → [i]	List all exported interface references in this name server.

Table 5.7 Services available from the nameserver module.

The listing then shows how f can be called through a remote operational binding (line 3 and 4, includes the creation of the binding).

The capsule is implemented as an Python process (address space) with one instance of the Capsule class managing this process. This instance of the Capsule is locally available through the local attribute of the capsule module. It implements a registry for its contained (and registered) components and a serving loop for remote access. The serving loop is implemented as a thread listing on a socket for incoming requests. One of the important tasks of the capsule is to generate global interface references. This is done by populating the interface reference with a capsule proxy representing itself (the capsule), the unique identifier for the component providing the interface, and the name of the interface (in the context of its component). An important feature of the serving loop is to catch all exceptions raised by remote request and forward them back (with the appropriate information) to the caller. This is important since uncatched exceptions in a remote capsule will terminate its process without any feedback to the caller.

5.2.2 Name server

A naming service provides a way of getting access to remote interfaces based on an address or a name. OOPP provides a simple naming service based on one or more *name servers*.

A name server is identified by a host identifier (IP address) and a port number and it is accessed through a name server proxy. A name server proxy is created from the NameServerProxy class when the host identifier and the (optional) port number are known.

Table 5.7 lists the services available from the nameserver module in the core package OOPP. There is a clear distinction between capsules and other interfaces in the name server since a capsule is not represented by an interface reference and the semantics of exports and imports are different.[2]

```
n = NameServerProxy(node,port)
u = capsule.local.registerComponent(a)
n.exportIRef("ai",capsule.local.getIRef(u,"i"))
```

(a) Export interface reference i

```
n = NameServerProxy(node,port)
j = n.importIRef("ai")
result = j.f(10,2)
```

(b) Import interface reference i

Listing 5.8 Using a name server to export and import interface reference i (named "ai" in the name server).

The exportCaps method registers the capsule p with the given name k in the name server. An importCaps method using name k will return a capsule proxy for the registered capsule if a capsule is registered using this name. Since an implicit binding now exists between the capsule proxy and its capsule[3] the capsule proxy can immediately be used to access its capsule.

The exportIRef method registers an interface reference i with the name k in the name server. This simple name server does not have a name hierarchy. One name server spans one name space for interfaces and one name space for capsules.

Two ways of importing an interface reference from the name server are available.

The lookupIRef method will return the interface reference of the given name. This interface reference can *not* be used to access the methods of the interface directly. An interface-lookup is usually used when an explicit binding later should be created to the exported interface. An example is an audio source exporting its audio output interface. An audio player (sink) with an audio input interface could look up (using the lookupIRef method of a name server proxy) the audio output interface and later create an explicit audio-stream binding between the audio output and the audio input interfaces. The audio-stream binding could be a user-implemented binding meeting the special needs of audio transfers. An lookupIRef method on the exported interface reference i of object a in Figure 4.1 will return the interface reference i. Listing 5.11 uses lookupIRef and then creates an explicit stream binding between the interfaces "src" and "i2".

The importIRef method will return an interface reference that matches the one exported. An implicit binding between the returned interface and the interface represented by the exported interface reference will automatically be created. The returned interface reference can immediately be used to invoke methods in the exported interface (and vice versa). An importIRef method on the exported interface i of object a in Figure 4.1 will return a new interface reference like j. And implicit (remote) binding will be created between interface i and j.

Listing 5.8 illustrates how interface i of component a is made available through a name server represented by the name server proxy n and later imported from the same name server. The implicit creation of a binding between interface i and j is hidden by the importIRef method.

NodeMngr() $\rightarrow$ o	Constructor of the NodeMngr class.
NodeMngrProxy() $\rightarrow$ o	Constructor of the NodeMngrProxy class.
o.ping() $\rightarrow$ k	Check if the node manager is there.
o.stopserve()	Terminate the node manager.
o.newport(k*) $\rightarrow$ v	Return (reserve) a new port number v. k is an optional string.
o.delport(v)	Release reserved port number v.

Table 5.8 Services available from the nodemngr module.

```
c = compositeFactory(
    {"ci": ("a","ai"), "co": ("b","bo")},
    {"comps": {"a": {"factory": componentFactory,
                     "args": (["ai", "ao"], A)},
               "b": {"factory": componentFactory,
                     "args": (["bi", "bo"], B)}},
     "iff": {"ao": ("a", "ao"), "bi": ("b", "bi")},
     "edges": [("ao","bi")]})
```

Listing 5.9 Use the factory compositeFactory to create the composite component from Figure 5.3.

5.2.3 Node manager

The *node manager* manages resources shared by different capsules on one node. In the current prototype this is TCP/IP and UDP/IP connection ports (port numbers). The existence of the node manager is hidden from the application programmer. All interactions with the node manager is done by the capsules. The capsule access the node manager through a node manager proxy. The first capsule started on a node will start the node manager when it discovers that the node manager is not there. The services provided from the node manager module in the core package of OOPP are listed in Table 5.8.

5.2.4 Factories

Several factories have already been briefly introduced above. These factories include the component factory, the generic composite component factory, and factories for different explicit bindings.

Listing 5.9 shows how the generic compositeFactory can be used to create a composite component similar to the one in Figure 5.3. The first argument specifies the external interfaces c_i and c_o (named "ci" and "co" in the code) and the second argument specifies the component graph. Component a and b are created with another factory (componentFactory) using their classes A and B. The main difference between the constructor of the Composite class and the compositeFactory is *when* the containing components of the composite component are created. A factory usually creates all the containing components while for

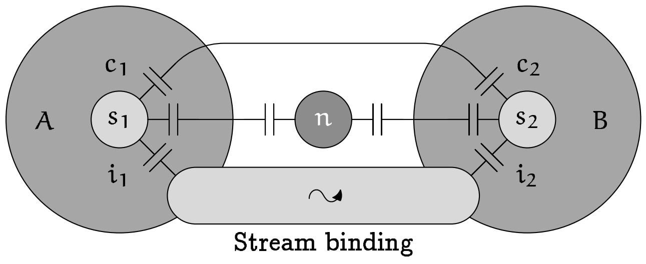


Figure 5.6 An audio source component s_1 in capsule A and an audio sink component s_2 in capsule B connected with a stream binding.

instance the Composite class constructor creates a composite component using containing components that already exists.

As seen in Listing 5.9 the syntax of the generic factory for composite components is complex. The reason is that it has to contain a complete description of the new composite component. The remoteBind, streamBind and sigBind functions presented above are examples of specialised factories with a simpler syntax.

5.2.5 Audio stream example

The example introduced here will evolve further in this and the following chapter as new concepts are introduced. The example contains an audio stream between an audio source and an audio sink.

Figure 5.6 gives an overview of the software architecture and the supporting infrastructure of the audio stream example. An audio source component has two interfaces, an audio output interface i_1 and a control interface c_1. The control interface exports the methods list, select, play and stop, and imports the method info. The audio output interface i_1 is bound with a stream binding to an audio input interface i_2 of an audio sink component. The audio sink component has two interfaces too. The first one is the audio input interface i_2 and the second one is a control interface c_2. The control interface imports the methods list, select, play and stop, and exports the method info. The control interface of the audio source is bound to the control interface of the audio sink with an implicit remote binding.

The list method provided by the audio source lists all the available audio sources. The select method selects one audio source to be played. The play method starts transferring the selected audio source from the audio source to the audio sink and the stop method stops the transfer. The info method provided by the audio sink can be used by the audio source to present information about the current (playing) audio source to its sink (typical the name of current source).

The stream binding transfers the audio from the source to the sink. Listing 5.10 and 5.11 describe the process of establishing this setup. The code in Listing 5.10 is performed

```
s1 = capsule.local.mkComponent(AudioSource)
n = NameServerProxy(node, port)
n.exportIRef("ctrl", capsule.local.getIRef(s1, "c1"))
n.exportIRef("src", capsule.local.getIRef(s1, "i1"))
```

Listing 5.10 Create the audio source s_1 in capsule A and export its interfaces. The audio sink is created and bound to the audio source in Listing 5.11.

```
s2 = capsule.local.mkComponent(AudioSink)
n = NameServerProxy(node, port)
c1 = ns.importIRef("ctrl")
i1 = ns.lookupIRef("src")
sb = streamBind(i1, capsule.local.getIRef(s2, "i2"))
```

Listing 5.11 Create the audio sink s_2 in capsule B, import the interfaces of the audio source and create a stream binding between the stream interfaces of the source and the sink. The audio source is created in Listing 5.10.

in capsule A and the code in Listing 5.11 is performed in capsule B. After these steps are performed the sink component can (through its control interface) list, select, play and stop the audio streams provided by the source component, and the source component can (through its control interface) present information about the source to the sink with the info method. The statement 'c1.play()' in capsule B starts transferring the currently selected audio source from the source s_1 to the sink s_2.

5.3 Meta-objects

The two meta-models provided by OOPP are implemented using encapsulation and composition meta-objects.

5.3.1 Encapsulation meta-objects

The encapsulation meta-model of a given object, interface or component is accessed through its *encapsulation meta-object*. The encapsulation function is used to get access to the encapsulation meta-object of an object, an interface or a component. The encapsulation meta-model is implemented in the encaps module of the meta package of OOPP.

Encapsulation meta-object services for objects

Table 5.9 lists the services provided by the encapsulation meta-object of objects. The inspect method returns a description of the object. The description contains the class, the

encapsulation(o) $\rightarrow e$	Fetch the encapsulation meta-object of object o.
e.inspect() $\rightarrow I$	Returns a description of the object.
e.addMethod(m, f, v^*)	Add (or replace) method m with the implementation f.
e.delMethod(m, v^*)	Delete method m from the object.
e.addPreMethod(m, f)	Add a pre-method f to method m.
e.delPreMethods(m, f)	Delete pre-method f (or all) from method m.
e.addPostMethod(m, f)	Add a post-method f to method m.
e.delPostMethods(m, f)	Delete post-method f (or all) from method m.
e.addWrapMethod(m, f)	Add a wrapper-method f to method m.
e.delWrapMethods(m, f)	Delete wrapper-method f (or all) from method m.
e.addGetAttr(k, f)	Add method f to be called when the value of attribute k is red.
e.delGetAttr(k, f)	Delete the method f (or all) of attribute k added with addGetAttr.
e.addSetAttr(k, f)	Add method f to be called when attribute k is given a new value.
e.delSetAttr(k, f)	Delete the method f (or all) of attribute k added with addSetAttr.
e.changeClass(C)	Change the class of the object to class C.
e.restore()	Restore the object.

Table 5.9 Encapsulation meta-object services available for objects (encaps module).

```
def f(self): return "OK"      1
e = encapsulation(o)          2
e.addMethod("m", f)           3
e.addMethod("n", f, 1)        4
e.delMethod("m")              5
e.delMethod("n", 1)           6
```

(a) Add and delete methods

```
def p(self,w): print "m called"      1
def q(self,w): print w["result"]     2
e = encapsulation(o)                 3
e.addPreMethod("m", p)               4
e.addPostMethod("m", q)              5
o.m()                                6
```

(b) Insert pre- and post-methods

Listing 5.12 Manipulate methods of an object.

attributes (attribute names), and the methods (method names) of the object.

Services used to add, replace and delete methods of an object are available. The addMethod method is used to add and replace methods of an object. The delMethod method removes methods from an object. In Listing 5.12 (a) a method named m with an implementation f is added to object o using the encapsulation meta-object e (line 3). Method n of object o is then replaced (overwritten) by a new method with the implementation f (line 4). The last optional argument of addMethod forces the replacement. Otherwise an exception would occur since method n already exists. Finally is method m and n removed from object o (line 5 and 6). The optional last argument of delMethod (line 6) enforce the complete deletion of method n. Otherwise, the old implementation of method n would become available after the deletion.

A set of services to add and delete pre-, post-, and wrap-methods are provided. Listing

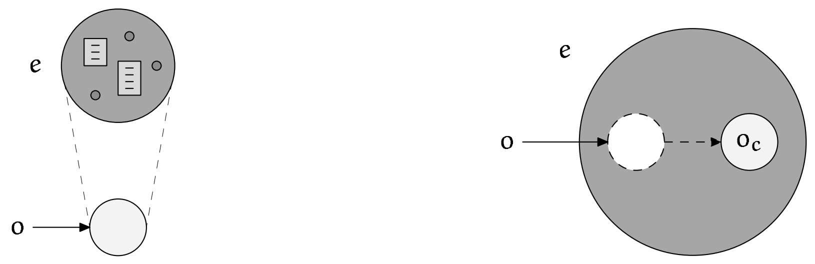
5.12 (b) illustrates how a pre-method p and a post-method q is added to method m of object o. Last line will call method m of object o. Before the actual method is called the message "m called" is printed (by pre-method p). After method m is called but before the call returns, the return value of m is printed (by post-method q). The first argument self of p and q is common with every method associated with an object. It is a reference to the object itself (like this in C++ and Java). The second argument w is a dictionary (map) containing information about the method call. This information includes a reference to the object, the name (key) of the method, a reference to the method implementation, the arguments of the method call, and a possible return value of the call (only available for post-methods and wrapping-methods after the actual call).

It is also possible to add meta-methods that will be called when attributes are read or modified. addGetAttr adds a meta-method to be called when the given attribute is read. addSetAttr adds a meta-method to be called when the given attribute i modified (given a new value). These added meta-methods can later be removed with delGetAttr and delSetAttr respectively.

The changeClass method can be used to change the complete implementation of the object. The object will become an instance of the new class. This service has to be used carefully since the new class can contain a new implementation not consistent with the current usage of the object. All instance attributes (the state) will be preserved. The common usage of this service is to change the class of the object to a super- or sub-class of its current class.

The last service listed in Table 5.9 will restore the object controlled by the encapsulation meta-object. The meaning of this is that the encapsulation releases its control over the object. In the current implementation all the implementation changes done on the object would disappear. The only change that will survive is a change of the class. The restore method will remove any overhead introduced in the usage of the object by the encapsulation meta-object.

The trick to make this work is to make a copy of the original object (its state) and then change the behaviour (implementation) of the original object. The original object becomes a shadow object while the actual state of the object is saved and updated in the copy o_c. Every access to the object now end up at the shadow object (the original object reference still points at the original object but it has changed its behaviour and is now called the shadow object). The meta-object controls the access to the object using this shadow object. Every non-manipulated method call or attribute change are forwarded to o_c. New methods (or replacements) added with addMethod from the encapsulation meta-object are installed in the shadow object with self referring to o_c. This ensures that the new methods are updating and accessing the state of the copy o_c. Figure 5.7 (a) shows the external (programmer's) view of an object o and its encapsulation meta-object e. Figure 5.7 (b) exposes the implementation details (o is a reference referring to the object). When the object is restored the shadow object are changed back to the class (implementation) of the original object and its state is restored (copied from o_c).



(a) Programmer's view **(b)** Implementation details

Figure 5.7 An object o and its encapsulation meta-object e.

```
def getx(self):
    return self.x
e = encapsulation(a)
e.addMethod("g",getx)
print a.g()
ei = encapsulation(i)
ei.addEMethod("g")
ej = encapsulation(j)
ej.addIMethod("g")
print j.g()
```

Listing 5.13 Adding methods to objects and interfaces.

Encapsulation meta-object services for interfaces

Table 5.10 lists the services provided by the encapsulation meta-object of interfaces. The inspect method returns a description of the interface. The description contains a reference to the object the interface is connected to (possible none) and information about the exported and imported methods.

The addEMethod and addIMethod methods are used to add an exported or an imported method to the given interface. Only the name of the new methods are necessary. The implementations should either be a part of an existing object or it should be added using the encapsulation meta-object of the object. Listing 5.13 illustrates how a method g with the implementation getx is added to object a, the exported methods of interface i and the imported methods of interface j (see page 58, Figure 5.1 and Listing 5.1 (a)). It is not necessary to re-bind the local binding between i and j. Method g will automatically become a part of this local binding.

Pre-, post- and wrap-methods can be added to exported and imported methods of interfaces too. In Figure 4.5 on page 50 these locations have the labels **2** and **3**. Listing 5.14 illustrates how a post-method with the implementation inc is added to method f of object a (line 11 and label **1**), to exported method f of interface i (line 12 and label **2**) and to imported method f of interface j (line 13 and label **3**). If method f of object a returns the value 1, then the first print statement of Listing 5.14 (line 7) will print "1" and the second

encapsulation(i) $\rightarrow e$	Fetch the encapsulation meta-object of interface i.
e.inspect() $\rightarrow I$	Returns a description of the interface.
e.add$X$Method(m)	Add a method with the name m to the exported or imported methods of the interface.
e.del$X$Method(m)	Delete exported or imported method with the name m from the interface.
e.addPre$X$Method(m, f)	Add a pre-method f to the exported or imported method with the name m.
e.delPre$X$Methods(m, f)	Delete pre-method f (or all) from the exported or imported method with the name m.
e.addPost$X$Method(m, f)	Add a post-method f to the exported or imported method with the name m.
e.delPost$X$Methods(m, f)	Delete post-method f (or all) from the exported or imported method with the name m.
e.addWrap$X$Method(m, f)	Add a wrap-method f to the exported or imported method with the name m.
e.delWrap$X$Methods(m, f)	Delete wrap-method f (or all) the from exported or imported method with the name m.
e.changeObject(o)	Change the object of the interface to object o.
e.restore()	Restore the interface.

Table 5.10 Encapsulation meta-object services available for interfaces (encaps module). X should be replaced by E for exported methods and I for imported methods.

(line 14) will print "1 2 3 4". When method f is called using interface j the second time (line 14), the return value will be printed and increased by 1 three times. The resulting return value 4 is then finally printed.

The changeObject service can be used to change the object the interface is connected (related) to. The restore method will remove the encapsulation meta-object of the interface.

The implementation of the encapsulation meta-object for interfaces is much simpler than the implementation of the encapsulation meta-object for objects. No shadow object is needed and all manipulation is done directly on the interfaces them self. The main reason for this is that the encapsulation meta-objects for interfaces have a limited scope. It is only meant to manipulate interfaces and the interface classes are created with these operations in mind. The encapsulation meta-objects for objects have to handle any instances of any classes and not only a limited set of OOPP classes developed in the OOPP project.

New (forwarding) methods added to interfaces using the addMethod method of encapsulation meta-objects for interfaces are added directly to the interface reference and its interface object (in the same way as these methods are added to the interface reference and its interface object when an interface reference is created and an interface is bound). These forwarding methods are implemented as a callable object. The result is that each forwarding method has its own state (storage). This is used to install (store) pre-, post- and wrap-methods of the exported and imported methods when addPre$X$Method, addPost$X$Method and addWrap$X$Method methods of the encapsulation meta-object for interfaces are called.

```
def inc(self,m):
    print m["result"],
    m["result"] += 1
a = A(); b = B()
i = IRef(a,["f"],[])
j = bindIRef(i,b)
print j.f()
ea = encapsulation(a)
ei = encapsulation(i)
ej = encapsulation(j)
ea.addPostMethod("f",inc)
ei.addEPostMethod("f",inc)
ej.addIPostMethod("f",inc)
print j.f()
```

Listing 5.14 Adding post-methods to object methods and interfaces.

encapsulation(c) $\rightarrow$ e	Fetch the encapsulation meta-object of component c.
e.inspect() $\rightarrow$ I	Returns a description of the component.
e.addIF(k, i)	Add (or replace) interface k with the implementation i.
e.delIF(k)	Delete interface k from the object.
e.changeObject(o_2) $\rightarrow$ o_1	Change the containing object o_1 of the component to object o_2.
e.replaceObject(o_2) $\rightarrow$ o_1	Replace the contained object o_1 and the interfaces of the component with the new object o_2 and its interfaces.
e.restore()	Restore the component.

Table 5.11 Encapsulation meta-object services available for components (encaps module).

Encapsulation meta-object services for components

Table 5.11 lists the services provided by the encapsulation meta-object of components. The inspect method returns a description of the component including its contained object (or component graph) and interfaces.

The addIF and delIF methods are used to add and remove interfaces from the component. The changeObject method actually replaces the contained object of the component. Be aware that the interfaces of the component are not changed. This could be done manually with addIF or by using replaceObject instead of changeObject. The replaceObject method replaces the contained object and all the interfaces of the component. When using replaceObject the new object has to contain interface reference attributes with names equal to the names of the existing interfaces of the component. The new interfaces of component will be the interfaces referred to by the interface reference attributes of the new object.

The contained objects o_a and o_b of components a and b from the example in Listing 5.3 on page 60 will be replaced with object o_c and o_d respectively. In Listing 5.15 (a) object o_a is replaced with object o_c using changeObject and addIF. In Listing 5.15 (b) object o_b is replaced with object o_d using replaceObject. Be aware that the replacement in Listing 5.15 (b) only works if object o_d has an interface reference attribute similar to j with a name equal to the name of the existing interface "bi" of the component.

The implementation of the encapsulation meta-objects for components are simply done

```
ea = encapsulation(a)
ea.changeObject(oc)
ea.addIF("ao", oc.ao)
```

(a) Using changeObject

```
eb = encapsulation(b)
eb.replaceObject(od)
```

(b) Using replaceObject

Listing 5.15 Replace the contained object and the interfaces of a component.

by accessing the attributes of the components directly. The attributes involved are the interface dictionary (mapping) attribute and the object reference (reference to the contained object).

5.3.2 Composition meta-object

The composition meta-object provides access to the component graph representing the composite component.

A composition meta-object is created with the composition function applied on an interface or a component. The returned composition meta-object can be used to expose and manipulate the component graph. Table 5.12 lists the most common services available from the composition meta-object [26].

Composition meta-object services

The inspect method returns a view of the component graph containing all components and all bindings of the composite component. This view can be used to get access to specific components or bindings, or to traverse the complete graph itself.

The bind method creates a new local binding between two interfaces in the component graph. The break method removes (breaks) a local binding between two interfaces in the component graph. These low-level methods manipulate the local bindings in the component graph directly. The insert, remove and replace methods are manipulating components and are expected to be used more often by the programmers.

The insert method inserts a new component in the component graph. The new component is bound to existing interfaces in the component graph as specified in the call. Existing bindings will be removed if any of the existing interfaces already are bound to other interfaces. This service can be used to insert a filter component in a flow. The optional argument specifies bindings to be established in the component graph. Such bindings are specified as a mapping from an interface name in the new component (k_2 in Table 5.12) to a name of an existing internal interface in the composite component (k_3 in Table 5.12).

The addIIF method adds a new internal interface ("iif") to the composite component. Later the bind method (see above) can be used to connect this interface to another internal

composition(c) $\rightarrow$ g	Fetch the composition meta-object of component c.
g.inspect() $\rightarrow$ I	Inspect (view) the component graph.
g.bind(k_1, k_2)	Connect interface k_1 and k_2 in the component graph.
g.break(k_1, k_2)	Break connection between interface k_1 and k_2.
g.insert(k_1, c, $\{k_2 : k_3\}^*$)	Insert (and bind) component c using the name k_1 in the component graph. The optional arguments specifies how the interfaces of the new component should be bound.
g.addIIF(k_1, (k_2, k_3))	Add internal interface k_3 from component named k_2. The new internal interface are named k_1.
g.remove(k)	Remove the contained component named k from the component graph (and from the composite component).
g.replace(c_2) $\rightarrow$ c_1	Replace component c_1 with component c_2. All bindings to interfaces of component c_1 is re-bound to interfaces of component c_2.

Table 5.12 Services provided by the comps module.

interface.

The remove method removes a component completely. Any bindings between interfaces of the removed component and other components in the graph will be removed.

The replace method is a mixture of the two operations above. It removes an existing component in the component graph and replaces it with a new one. Existing bindings to the original component are removed and new bindings are created to the new one. The new and the original component must be similar (including equal interfaces). The new component inherits the role of the original component.

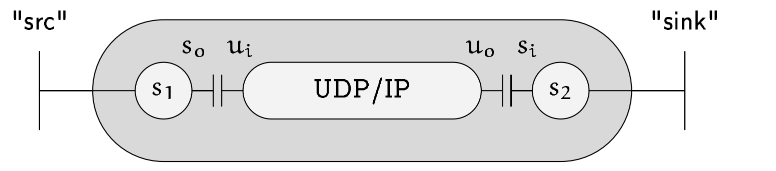
The composition meta-object is implemented by accessing the composite component directly. The composite component class has been implemented with this in mind. The internal representation of the component graph of a composite component are made in such a way that the composition meta-object services above can be implemented efficiently.

Examples of the usage of the composition meta-object

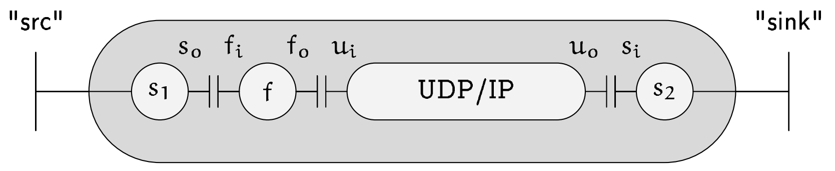
An example from Section 4.4.2 in Chapter 4 included an MPEG encoder and decoder connected with stream binding s (see also Figure 5.8 (a)). The composition meta-object c of s is fetched and used to insert a filter component f into the source side of the binding (see Figure 5.8 (b)). Listing 5.16 (a) contains the code used to insert the filter. The f_i and f_o interface of component f is automatically added to the internal interfaces of component s (the binding) when the insert method is used like this. Listing 5.16 (b) illustrates how this could be done manually (not using the optional argument to insert).

5.3.3 Audio stream example continued

This example started at page 72 in Section 5.2.5 above. We will now extend it by exposing some of the meta-models in the example.



(a) Without filter



(b) With filter f

Figure 5.8 Stream binding s with or without a filter component.

```
g = composition(c)
g.insert("f", f,
     {"fi":"so", "fo":"ui"})
```

(a) Using one insert statement

```
g.insert("f", f)
g.addIIF("fi", ("f","fi"))
g.addIIF("fo", ("f","fo"))
g.bind("fi", "so")
g.bind("fo", "ui")
```

(b) Manually connecting graph

Listing 5.16 Insert component f in the component graph.

The audio source s_1 produces a frame of audio samples in a timely manner. The size of each frame is 3840 bits (or 480 byte). With a sample frequency at 16kHz and 8 bits per sample each frame will contain 20ms of audio (50 frames per second).

The audio source component s_1 has a timed thread that is scheduled to produce a new frame every 20ms. The audio sink s_2 plays an audio frame every time one is received at the input interface. The stream binding is a composite component containing a source stub m_1, a sink stub m_2, and an UDP/IP binding between m_1 and m_2. The source stub m_1 transfers the audio frames received from the audio source as quickly as it can over the UDP/IP binding to the sink stub m_2. The sink stub forwards the audio frames to the audio sink as soon as they arrive. Figure 5.9 illustrates this setup with the components contained in the stream binding exposed. Figure 5.6 gives a complete view of the setup.

Audio is very sensitive to time. Each sample must be played exactly the given period of

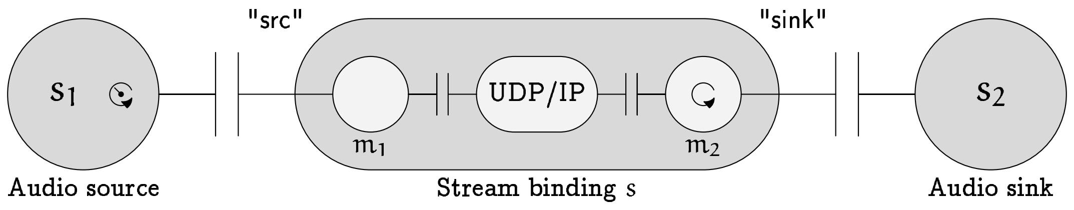


Figure 5.9 An audio source and an audio sink bound with a stream binding.

```
e = encapsulation(s)
e.addIF("sink", b.interface["bo"])
g = composition(s)
g.addIIF("mo", ("m2","sink"))
g.insert("b", b, {"bi":"mo"})
```

Listing 5.17 Add a buffer component b to the stream binding s.

time after the previous one. This is difficult to achieve when the audio is transfered over a network in small audio frames. The next frame must be available imitately after the last sample in the current frame has been played. One way to achieve this is to always have a set of pre-fetched frames available. We usually do not want to pre-fetch too many audio frames, since this increases the latency (the time from the audio generated at the source to it is played at the sink). A buffer component b is added to the stream binding s to store and (timely) forward the pre-fetched audio frames to the audio sink. This is done through the composition meta-object of the stream binding. The composition meta-object g are used to insert the new buffer component b into the component graph of s. The encapsulation meta-object e is used to change the external "sink" interface of s. Listing 5.17 shows how this is done. Figure 5.10 illustrates the resulting system.

Notes

1. Python and information about Python is available from http://www.python.org/.

2. An implicit binding exists between a name server and a name server proxy. This binding and other bindings between infrastructure components and their proxies are not related to implicit bindings (binding objects) between interfaces represented by interface references. The reason is that this sort of binding belongs to the infrastructure and is used to implement programming structures (including higher-level bindings) provided to the application programmers. They are lower-level building blocks and they can not be build from programming structures they are partly implementing.

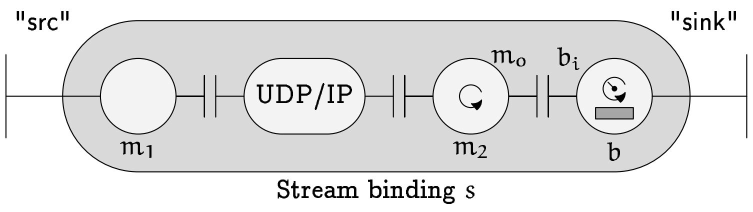


Figure 5.10 Introducing a buffer component in the stream binding from Figure 5.9.

3. All proxies (capsule proxies, name server proxies and node manager proxies) use a low level message protocol supporting announcements (a message without a reply), synchronous messages (send a request and wait for the reply) and asynchronous messages (send a request and pick up the reply later). An implicit binding based on this low level message protocol exists between a proxy and the object it represents as soon as the proxy is created. A binding process is not necessary before the proxy is used.

Chapter 6

Quality of service management

Management is the task of establishing and maintaining key-properties of a system. The following will focus on dynamic quality of service (QoS) management. *Dynamic QoS management* includes monitoring, adaption, re-negotiation, policing and maintenance of QoS properties. Typical *static QoS management* functions are admission control, QoS negotiation and resource reservation. The static QoS management functions will be ignored in the following.

6.1 Management roles

The proposed management pattern consists of a set of *management objects* with different roles. Several management objects can have the same role, but one management object can also fill more than one role. The three different management roles *monitoring*, *strategy selecting* and *strategy activating* have been identified. Figure 6.1 illustrate the different roles in an adaptive producer-consumer setup. The distinct roles are selected to separate conceptual different tasks in a management setup. Such separation of concern makes it possible to separately develop, maintain and reuse different management tasks. A description of the tasks of each role are described below.

6.1.1 Monitor

A management object with a monitoring role is simply called a monitor. A *monitor* is used to collect information from the activities of components and objects in a running system. It can perform some simple filtering of the collected information to highlight interesting events and situations. It is also possible to have several levels of monitors where the next level monitor collects information from lower level monitors. In a running system without any observed problems (interesting events or situations) the monitors will collect information without reporting it any further to other management objects.

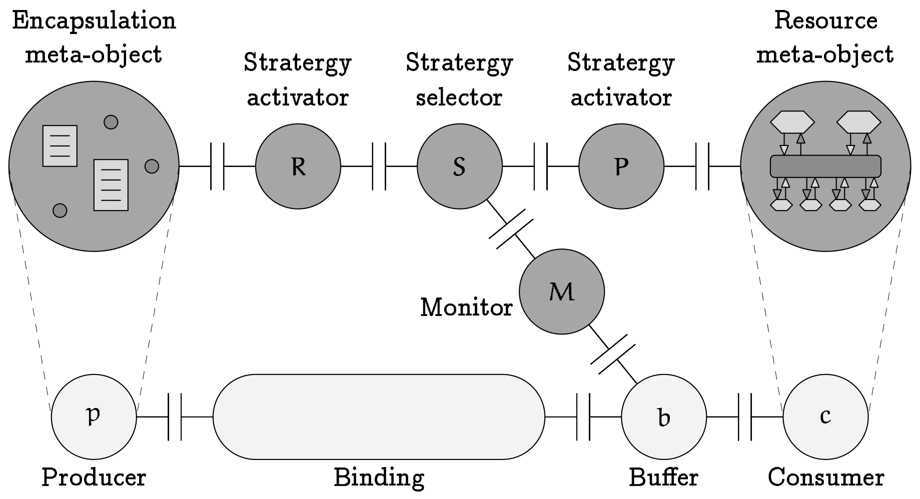


Figure 6.1 The different roles of management objects in an adaptive producer-consumer setup.

A monitor can either use services from the observed component or reflection to expose activities related to a given component. For example, a buffer component can provide an interface where the monitor can register for notification of buffer overflows. The monitor will then receive a signal every time a buffer overflow occur. A buffer component that does not provide the notification of overflow service can be monitored for overflows using reflection.

For example, a buffer component b has an addItem method that adds a new item to its buffer. First, the encapsulation meta-object e_b of the buffer component is fetched using the encapsulation function. Then, addPreMethod of e_b is used to add f as a pre-method to the addItem method of the buffer component b. The f method check if the buffer is full before the addItem method is performed, and sends a signal (to the monitor) if this is true. This signal indicates that addItem will generate a buffer overflow.

The buffer component contains received requests that will be consumed (processed) by a local process. It is acceptable to loose a request now and then in a buffer overflow since it will be retransmitted after a given timeout. But it is not acceptable if we loose too many requests to often. This indicates that the requests are consumed (processed) slower than they are received and a solution is either faster processing or fewer (slower) requests. The monitor receives 'overflow' signals each time an overflow occurs in the buffer component. It keeps a counter c of the number of received 'overflow' signals in a given period T. It produces an 'oto' (overflow-to-often) signal if the counter c exceeds a given level n during a period T. This 'oto' signal is typically received by a management object with a strategy selecting role.

6.1.2 Strategy selector

A management object with a strategy selecting role is called a strategy selector. A *strategy selector* receives interesting events from monitors and makes decisions based on that and its internal state. Its internal state is influenced by previous received events and current time. A decision can be to do nothing, to change its internal state, to set or reset timers, or to send a signal to a selected receiver (or a combination of the last three).

A strategy selector waits for 'oto' (overflow-to-often) signals from the monitor described above. When the first 'oto' signal is received the strategy selector produces a 'ip' (increase-processing) signal. It then waits (ignores 'oto' signals) for a given period of time to be assured that the system has stabilised with the increased processing power to the consumer of the request. It continues like this every time it receives an 'oto' signal until a 'mp' (maximum-processing) signal is received. This signal indicates that it is not possible to increase the processing power of the consumer any more, and a different strategy must be selected. From now on every 'oto' signals will produce a 'lrr' (lower-request-rate) signal followed by a given period of time when 'oto' signals are ignored. The intention of the 'lrr' signal is to instruct the producer of the requests to slow down.

6.1.3 Strategy activator

A management object with a strategy activating role is called a strategy activator. A *strategy activator* is specialised to activate its given strategy. This may include the process of deactivating the old strategy, building up the necessary structure (and infrastructure) of the new strategy, and activating the new strategy. Strategy activators can do anything from tweaking some attributes in a component in a running system to completely remove an rebuild big chunks of the structure and infrastructure of a complete system.

The 'ip' (increase-processing) and 'lrr' (lower-request-rate) signal from the example above are received by two different strategy activators.

The strategy activator P receiving 'ip' signals will instruct the consumer of the request to speed up. This can be done by accessing the resource meta-model of the consumer component. The resource meta-model can be used to increase the periodic time-slot available to the thread processing the requests or increase the priority of the thread (depending on the policies available). P will send a 'mp' (maximum-processing) signal back to the strategy selector when it can not increase the processing power any further.

The strategy activator R receiving 'lrr' signals will instruct the producer of the requests to slow down. It can do this by accessing the encapsulation meta-object e_p of the producer. The producer has a produce-request loop that forwards the requests to the buffer. This loop includes the produceRequest method performing this action (including calling the addItem method of the buffer b). R can use the addPreMethod service of e_p to insert a method s to be executed before the produceRequest method. The produceRequest method delivers a request to the buffer component (over the binding). The pre-method s sleeps a given period of time before it returns. This will slow down the produce-request loop in the producer.

6.2 Management objects

The nature of the different demands on the managements objects has resulted in two different approaches in the realisation of them. The way monitors and strategy selectors behave is well suited to describe with an automaton, and especially with a *timed automaton*. Therefore, the monitors and strategy selectors are realised as running timed automata. The way strategy activators behave varies a lot. Some strategy activators only tweak some arguments in an attribute of a component in the system, and some removes and rebuilds almost the complete structure of a system. The approach used to realise strategy activators strongly depends on its given tasks, and they can be anything from a simple object to some sort of scripting languages. One common approach is to access one or more of the meta-models of the components in a system and change the structure and behaviour of the components involved.

6.2.1 Automata

As discussed in [32], timed automata [3, 4] have been shown to be useful for modelling (management) systems. Automata are easy to understand and they have a large number of validation and verification techniques associated with them. Also, a large number of tools and simulators are available and can be used when their behaviour and properties are to be understood. Timed automata are modelled with time related behaviour. This is important in implementing QoS management functions. Such functions are often related to time and time constraints.

Another nice thing with automata is that they fit the OOPP programming model very well. An automaton can be wrapped as a component with its behaviour described in a timed automaton description. Output and input signals are modelled as source and sink signal interfaces, respectively. Such interface pairs are connected with signal bindings.

An *automaton* has a given set of states with edges (transitions) between them. One state is called the initial state, and this is the state where the automaton starts (drawn with two circles like state S_0 in Figure 6.2). A state can have a guard. A guard is a conditional expression that is either true or false. The automaton is not allowed to be in a state with a false guard (it must change state). To change state is has to select an edge starting in the current state. The new state of the automaton is the state where the selected edge ends. If the state is changed because the current state has a false guard it has to select a non-named edge (an edge not marked with an incoming signal like 'overflow?'). The edges can also contain guards. Only edges with true guards can be selected. If the automaton receives an input signal (like 'overflow'), an edge with its given name (and a true guard) is selected. An edge can also contain an output signal (like 'oto!'). The output signal is sent

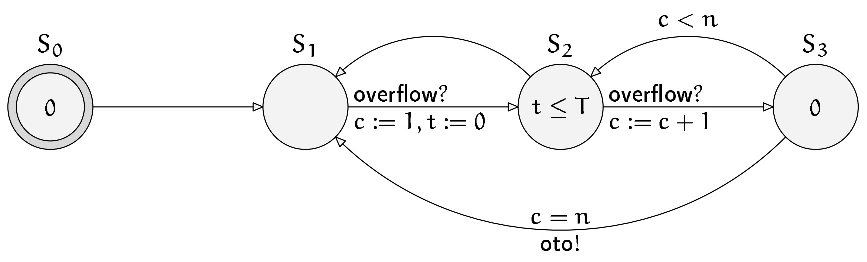


Figure 6.2 The automaton description of the monitor M in the producer-consumer management setup (Figure 6.1). The automaton receives 'overflow' signals and produces 'oto' (overflow-to-often) signals. Value c is a counter, value t is a timer and values n and T are constants.

every time this edge is selected. In an automaton description (like the on in Figure 6.2), incoming signals are marked with a question mark (like 'overflow?') and output signals are marked with an exclamation mark (like 'oto!'). Statements used to set the value of a variable (counter) or a timer use the assignment operator ':=' (like $t := 0$). Guards use test operators to check if a guard is true or false. The following test operators are used: '$=$', '$>$', '$\geq$', '$<$' and '$\leq$'. A false guard is specified with the number 0.

In the following, a simplified notation for automata is used. The reason is to not overload the automata descriptions (make them more readable). The description of their behaviour will be unambiguous if the interpretation described below is used. In standard automata descriptions a non-labeled edge without a guard or with a true guard can be selected even if the guard of the current state is true (or it is without a guard). However, in the simplified notation used here, every non-labeled edge from a state with a guard has a guard that is the opposite of the guard of its source (start-point) state. This is implicit and not specified in the description. In other words, an automaton does not select an edge before it has to do it (and that is when its current state has a false guard or a signal is received). For example, the edge from S_2 to S_1 in Figure 6.2 has an implicit (non-specified) guard $t \geq T$.

Example

Figure 6.2 is an example of an automaton that monitors 'overflow' signals and produce a 'oto' (overflow to often) signal when it receives the 'overflow' signals more than n times in a time-period T. The automaton has four states S_0, S_1, S_2 and S_3, a counter c, and a timer t. The constant T is the period 'overflow' signals are counted and n is the number of 'overflow' signals in period T that is needed to produce a 'oto' (overflow to often) signal. S_0, and S_3 have a false (0) guard. The automaton is not allowed to be in a state with a false guard, and as soon as it arrives one of these it has to move on to the next state. It selects an edge from the set of non-named edges with true guards. State S_2 has a timer guard where the guard will be false after T seconds. The edges out of S_3 have guards. The edge from S_3 to S_1 can only be selected if $c = n$, and the edge from S_3 to S_2 can only be selected if $c < n$. The edge from S_3 to S_1 produces the output signal 'oto' (overflow to often). The resulting automaton behaves like described in section 6.1.1 above.

In a management setup this automaton would be wrapped an OOPP component with two signal interfaces. One incoming interface for 'overflow' signals and one outgoing interface for 'oto' signals.

6.2.2 Other management objects

Management objects not described and implemented as an automaton could be any type of objects or components. They could even be specialised programs or scripts that start performing their tasks when a signal i received. A strategy activator is typically such a management object.

Sometimes the components of the managed system provide control interfaces used to change their behaviour. A typical example is an audio source that has a control interface that can be used to change the quality of the presented media. The control interface can be used to select between CD-quality (stereo, 44100 16 bits samples per second) or phone-quality (mono, 8000 8 bits samples per second). A strategy activator can then be implemented to use this control interface to lower the quality of the media if the current high quality presentation is impossible to achieve.

A common approach is to use strategy activators accessing the meta-models of the objects involved. The encapsulation meta-model can be used to tweak some attributes or implementation details of an object, to add or remove exported and imported methods of an interface, or to add or remove interfaces of a component. The composition meta-model can be used to manipulate composite components like bindings. New components (like filters and buffers) can be added, existing contained components can be replaced, and the complete structure (component graph) can be restructured.

6.3 Performing the management functions

Figure 6.1 is an example of a management setup with a monitor M, a strategy selector S and two strategy activators P and R. A description of the different roles in this example was given above.

6.3.1 Monitoring with automata

The monitoring task can be very simple, and in some cases the automaton can be left out. If the monitoring task is to generate a signal every time a given method in a given component is called the only needed action is to add a pre-method to this method. This pre-method will generate a signal every time it is called. But usually the monitoring task is a little bit more sophisticated. Figure 6.2 is an example of a more sophisticated monitoring task. It is an automaton description of monitor M from Figure 6.1. The monitor is in state S_1 as long as no 'overflow' signals are received. It moves to state S_2 when the first 'overflow' signal occurs. The counter c now counts the number of received 'overflow' signals in a period of length p. This is done by the S_2–S_3–S_2 loop. The automaton moves to state S_1 and produces an 'oto' signal if c reaches n during this period. The automaton moves to state S_1 without producing any signals when the period is over and less than n 'overflow' signals were counted.

As described above, M receives 'overflow' signals and generates 'oto' (overflow-to-often) signals. M is bound to a notification interface of the buffer it monitors with a signal binding. This signal binding transfers the 'overflow' signals to M. M is also bound to strategy selector S with a signal binding. This signal binding transfers the 'oto' signals from M to S.

6.3.2 Strategy selecting with automata

Figure 6.3 is an example of a strategy selector. It is an automaton description of strategy selector S from Figure 6.1.

The strategy selector is in state S_1 as long as no 'oto' (overflow-to-often) and no 'mp' (maximum-processing) is received. In this state each 'oto' signal will produce an 'ip' (increase-processing) signal (the S_1–S_2–S_1 loop). New 'oto' signals are ignored for a period p after each 'oto' signal (to wait for the system to stabilise after an output signal). The automaton moves to state S_3 when an 'mp' signal is received. In this state each 'oto' signal will produce an 'lrr' (lower-request-rate) signal (the S_3–S_4–S_3 loop). New 'oto' signals are ignored for a period p after each 'oto' signal.

S receives 'oto' signals and generates either 'ip' or 'lrr' signals. The 'oto' signals are received from the monitor M over a signal binding. S also has a signal binding to strategy selector P where 'ip' signals are sent and a signal binding to strategy selector R where 'lrr' signals are sent. It can also receives 'mp' signals from P.

6.3.3 Strategy activating

P and R from Figure 6.1 are strategy activators. P has access to the resource meta-object of the consumer and uses this meta-object to increase the processing power of the consumer. R has access to the encapsulation meta-object of the producer and uses this meta-object to slow down the producer.

6.3.4 Meta-management

Meta-managing is the process of managing the management components. A set of management objects tries to maintain the quality of service following a given policy or strategy. The meta-management process can monitor how well this strategy works and decide to

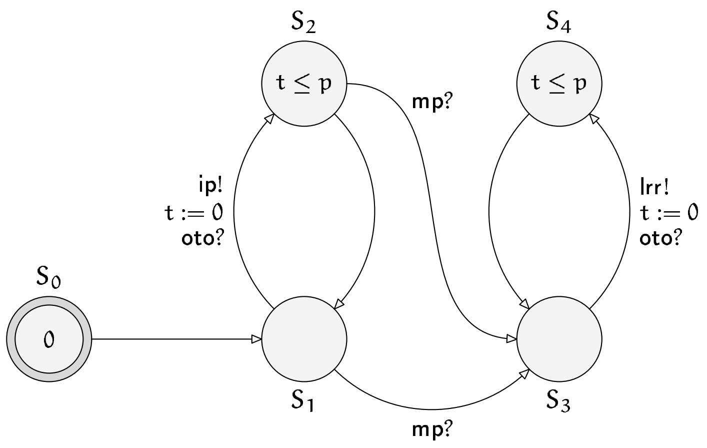


Figure 6.3 The automaton description of the strategy selector S in the producer-consumer management setup (Figure 6.1). The automaton receives 'oto' (overflow-to-often) and 'mp' (maximum-processing) signals and produces 'ip' (increase-processing) and 'lrr' (lower-request-rate) signals. The value t is a timer and p is a constant.

Automata(f^*, x^*) $\rightarrow$ o	Constructor of the Automata class. The optional argument f is the method used when an event is produced by the automaton. The optional argument x is the automata description.
o.install(x)	Installs the automata description x in the automaton o.
o.printState()	Print current state to standard output.
o.run()	Initialise and start the automaton.
o.stop()	Stop the automaton. New events will now produce an exception.
o.newEvent(x)	A new input event (named x) for the automaton.

Table 6.1 Services available from the automata module.

switch to a completely different strategy if the first one fails. The meta-management process then replace the existing management setup with one that implements the new strategy.

6.4 QoS management in OOPP

QoS management in OOPP is implemented in the mngt package. This package contains a set of modules used to implement, configure and deploy automata.

6.4.1 OOPP automata

Table 6.1 lists the services provided by the automata module of the OOPP mngt package. This module implements the Automata class used to create instances of timed automata.

```
nets 3
   hook"main" > 0
   struct"OTO monitor"
   net 2
      structs 1
         :0 "Config"
      behavs 1
         :0 "int c, T, n; clock t"
      struct 0
      behav 0
      hook "synch_vector"
   net 1
      structs 4
         :0 "S0,false"
         :1 "S1"
         :2 "S2,t<=T"
         :3 "S3,false"
      behavs 6
         :0 "n:=2,T:=8"
         :1 "overflow?,c:=1,t:=0"
         :2 ""
         :3 "overflow?,c:=c+1"
         :4 "c<n"
         :5 "c=n,oto!"
      logic "initial">0
      hook "automaton"
      vertice 4
         vertex0
            struct 0
            edges 1
               edge0
               behav 0
                -> 1
         vertex1
            struct 1
            edges 1
               edge0
               behav 1
                -> 2
         vertex2
            struct 2
            edges 2
               edge0
               behav 2
                -> 1
               edge1
               behav 3
                -> 3
         vertex3
            struct 3
            edges 2
               edge0
               behav 4
                -> 2
               edge1
               behav 5
                -> 1
   net 0
      struct _< 1,2
      hook "synch_vector"
```

Listing 6.1 FC2 definition of the monitor from Figure 6.2 (using $n := 2$ and $T := 8$).

An automaton is initialised with an output event method and an FC2 [77] based description of a timed automaton. The output event method has one argument, the name of the produced event. The FC2 description has to be parsed by the FC2 parser in the fc2 module of the OOPP mngt package. An automaton receives input events through its newEvent method. The automaton can be started and stopped using the run and stop methods respectively. A printState method is provided to print the current state to standard output.

Listing 6.1 shows the FC2 description of the monitor described in Figure 6.2. The description is split into three different parts (nets). One (net1, starts at line 12) contains the actual automaton description. The two other parts (net0 and net2) contain declarations. Four different states (see vertice line 27) are specified. Each state has a name and an optional guard (see structs line 13). S_0 and S_3 have false guards. S_2 has the guard $t \leq T$. S_0 (vertex0 line 28) has one edge ending at S_1. This edge initialises the constants n and T (behav 0 line 19). S_1 (vertex1 line 34) has one edge ending at S_2. This edge has the label (input signal) 'overflow? and initialises the counter c and the timer t (behav 1 line 20). S_2 (vertex2 line 40) has two edges ending at S_1 and S_3 respectively. The first edge has the (implicit) guard $t \geq T$ (behav 2 line 21). The second edge has the label (input signal) 'overflow?' and adds one to the counter c (behav 3 line 22). S_3 (vertex3 line 49) has two edges ending at S_2 and S_1 respectively. The first edge has the guard $t < n$ (behav 4 line 23). The second edge has the guard $t = n$ and the label (output signal) 'oto!' (behav 5 line 24).

The behaviour of an automaton is described in the FC2 format. In the automata objects this information is kept in a way that makes the running of the automata efficient. Time is introduced in a timed automaton using timers. A special class Values has been made for keeping all the variables (timers and counters) of an automaton. This class makes the values of timers correct by relating them to the actual time when they are read and written. The value of a timer will change dependent on the time. For example, if the timer t is given the value O at a given time, the value of the timer will be 5 five seconds later. In a timed automaton timers plays an import roles in the guards. Guards of edges containing timers will work properly since the value of timers are fetched from an instance of the Values class. As a result of our simplified notation, guards on edges are only tested when current state becomes (is) false or when input signals are received. However, guards on states might change from true to false and have an impact on the running automaton when there are no other activities. To implement such a behaviour, a new thread is created when the automaton moves to a state with a guard containing timers. This guard expression is analysed and the thread sleeps until it believes[1] the guard might change to false. When the thread wakes up it checks the state of the guard. If the guard is still true, the thread sleeps again until it believes (based on a new analyse) the guard might change to false.

AmComp(x*) → c	Constructor of the AmComp class. The optional argument x is the automata description.
c.interfaces	The interfaces of an AmComp component.
c.interfaces["a_ctrl"]	The control interface. The control interface exports most of the public methods of the wrapped automaton (an instance of the Automata class).
c.interfaces["a_in_*X*"]	The input interface "*X*".
c.interfaces["a_out_*X*"]	The output interface "*X*".

Table 6.2 Services available from the amcomp module.

```
# Create monitor and a control interface
M = AmComp(FC2(open("M.fc2")).fc2py)
m = bindIRef(M.interfaces["a_ctrl"])

# Create strategy selector and a control interface
S = AmComp(FC2(open("S.fc2")).fc2py)
s = bindIRef(S.interfaces["a_ctrl"])

# Connect both automata to buffer and strategy activators
localBind(b.interfaces["overflow"],M.interfaces["a_in_overflow"])
localBind(M.interfaces["a_out_oto"],S.interfaces["a_in_oto"])
localBind(S.interfaces["a_out_ip"],P.interfaces["ip"])
localBind(S.interfaces["a_out_lrr"],R.interfaces["lrr"])

# Start automata
m.run()
s.run()
```

Listing 6.2 Create, connect and start the monitor M and the strategy selector S from the management setup in Figure 6.1.

6.4.2 OOPP automata components

The amcomp module in the OOPP mngt package provides a component class wrapping timed automata (instances of the Automata class). Table 6.2 lists the services of the amcomp module. The monitors and the strategy selectors in a management setup in OOPP are instances of the AmComp class connected with signal bindings. Such components has one control interface (named "a_ctrl") and one signal interface for each incoming and outgoing signal of the automaton. The monitor from Figure 6.2 will have a control interface "a_ctrl", an incoming signal interface "a_in_overflow", and an outgoing signal interface "a_out_oto".

Listing 6.2 contains the code used to create, connect and start the monitor M and the strategy selector S from the example in Figure 6.1. Line 2 and 3 creates monitor M from the automaton description in the file M.fc2. The control interface of M is made available through interface reference m. Line 6 and 7 creates strategy selector S from the automaton description in the file S.fc2. The control interface of S is made available through interface reference s. M is connected to the buffer component b (line 10) and the strategy selector S (line 11). S is connected with strategy activators P (line 12) and R (line 13). Finally, the two automata M and S are started using their control interfaces (line 16 and 17).

6.4.3 Audio stream example continued

The audio stream example from page 72 in Section 5.2.5 and page 80 in Section 5.3.3 will now be extended with a management setup. This management setup is illustrated in Figure 6.4.

The audio sink produces 'unsync' signals when it fails to play an audio sample because it is not yet available. The monitor M monitors these signals and produces an 'uto' (unsync-to-often) signals if this happens to often. M is similar to the monitor specification in Figure 6.2 (if the 'overflow' input signals are replaced with 'unsync' input signals, and the 'oto' output signal is replaced with an 'uto' output signal). However, the monitor from Figure 6.2 counts in fixed time-periods of length T. A burst of 'overflow' signals overlapping the end of one period and the beginning of the next period could be missed even if more than n 'overflow' was received in T seconds. Figure 6.5 implements a monitor with a 'floating' time period. This automaton are counting received 'unsync' signals in the S_1–S_2–S_1 loop. If the counter c is greater than zero it is decreased by one every p seconds. $p = T/n$ and 'uto' are generated if more than n 'unsync' signals are received in a period of T seconds. Both S_1–S_1 loops reset the timer t to start counting for a new period p. If more than n 'unsync' signals are received in a period T then the S_2–S_3 edge is selected and an 'uto!' signal is produced. The automaton stays in state S_3 waiting for the system to stabilise after sending the 'uto! signal (waits for q seconds). It then resets the counter and timer and starts counting again (S_3–S_0–S_1). The values of the constants n, p and q have to be tuned for this monitoring task.

The 'uto' signals are received by strategy selector S. The first 'uto' signal will produce

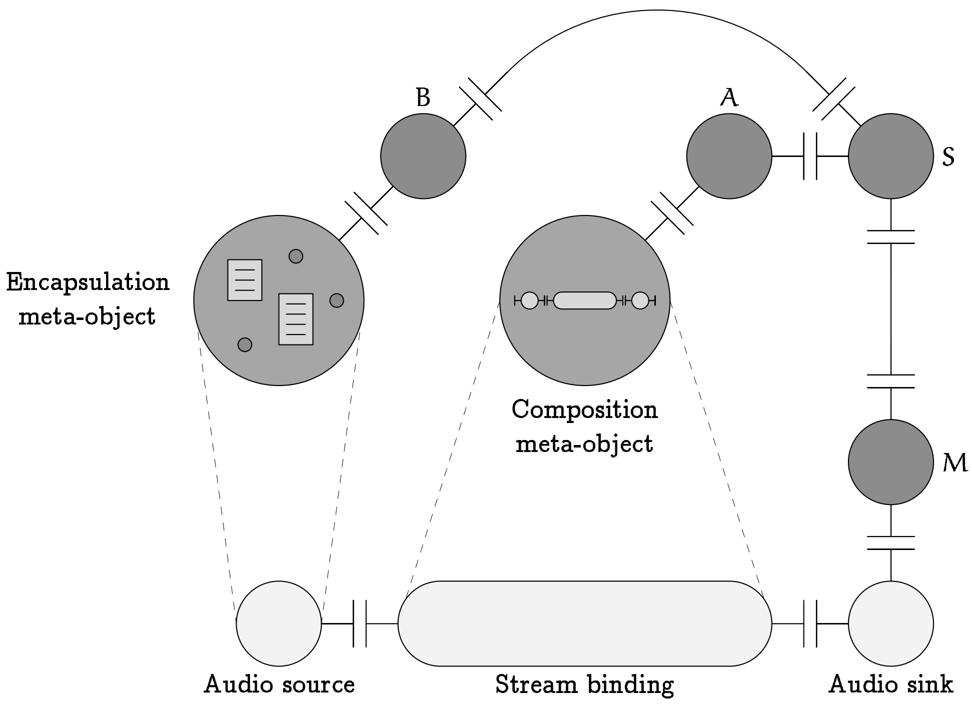


Figure 6.4 An audio stream management setup with a monitor M, a strategy selector S, and two strategy activators A and B. The strategy activators manipulate the meta-objects of the audio source and the stream binding.

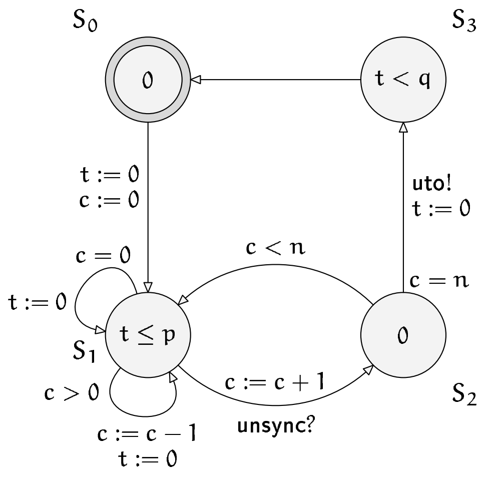


Figure 6.5 The automaton description of the monitor in the audio stream example in Figure 6.4. The automaton receives 'unsync' signals and produces 'uto' (unsync-to-often) signals. Value c is a counter, value t is a timer, and values n, p and q are constants.

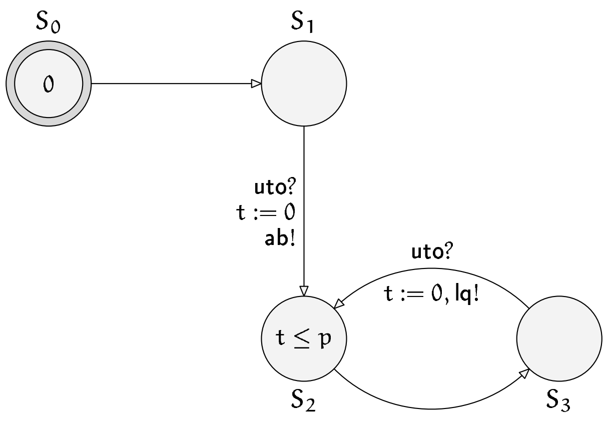


Figure 6.6 The automaton description of the strategy selector S in the audio stream management setup (Figure 6.4). The automaton receives 'uto' (unsynchronised to often) signals and produces 'ab' (add buffer) and 'lq' (lower quality) signals. The value t is a timer and the constant p is the period to wait for the system to stabilise after and output signal (a period when input signals are ignored).

an 'ab' (add-buffer) signal. Later 'uto' signals will produce 'lq' (lower-quality) signals. The strategy selector will ignore input signals for a period p after it has produced an output signal (to wait for the system to stabilise after an output signal). An automaton description of strategy selector S is given in Figure 6.6.

The 'ab' signal from strategy selector S is received by strategy activator A. A has access to the composition meta-object of the stream binding. When the 'ab' signal is received A uses the composition meta-object to insert a buffer component in the stream binding (see Figure 5.10 at page 83).

The 'lq' signal from strategy selector S is received by strategy activator B. B has access to the encapsulation meta-object of the audio source. When a 'lq' signal is received B uses the encapsulation meta-object to lower the quality of (and the amount of data from) the audio source being played.

Listing 6.3 contains the code used to create the management setup described above. This code is executed in capsule A (see Figure 5.6). The AddBuffer (line 2) and LowerQuality (line 11) classes implement the strategy activators A and B, respectively. The local capsule is made available through pA and the remote capsule is made available through pB (sing the name-server n).

The monitor M and strategy selector S are created using the services of their capsules. Access to their control interfaces are also made available (line 24–27). The buffer and monitor are connected with a local binding (line 30). The monitor and strategy selector are connected with a signal binding (line 31). Then, the strategy activators are created and connected with the strategy selector (line 34–37). Finally, the two automata are started using their control interfaces.

```
# Strategy activator class for activator A
class AddBuffer:
    def __init__(self, s):
        self.s = s
        self.ab = SigSink(self)
    def event(self):
        g = composition(self.s)
        # Add buffer using the composition meta-object

# Strategy activator class for activator B
class LowerQuality:
    def __init__(self, s):
        self.s = s
        self.lq = SigSink(self)
    def event(self):
        e = encapsulation(self.s)
        # Lower quality using the encapsulation meta-object

# A is the local capsule and B is the remote capsule
pA = capsule.local
pB = n.importCapsule("Capsule B")

# Create strategy selector, monitor and their control interfaces
M = pB.mkComponent(AmComp, (FC2(open("M.fc2")).fc2py,))
m = bindIRef(pB.getIRef(M, "a_ctrl"))
S = pA.mkComponent(AmComp, (FC2(open("S.fc2")).fc2py,))
s = bindIRef(pA.getIRef(S, "a_ctrl"))

# Connect monitor to source and strategy selector
localBind(pB.getIRef(s2, "unsync"), pB.getIRef(M, "a_in_unsync"))
b = sigBind(pA.getIRef(S, "a_in_uto"), pB.getIRef(M, "a_out_uto"))

# Create strategy activators and connect them to the strategy selector
A = pA.mkComponent(componentFactory, (["ab"], AddBuffer, (s,)))
B = pA.mkComponent(componentFactory, (["lq"], LowerQuality, (s1,)))
localBind(pA.getIref(S, "a_out_ab"), pA.getIRef(A, "ab"))
localBind(pA.getIref(S, "a_out_lq"), pA.getIRef(B, "lq"))

# Start automata m (monitor) and s (strategy selector)
m.run(); s.run()
```

Listing 6.3 Implementation of the audio stream management setup.

Notes

1. In the current implementation this enforces some limitations on how complex the guard expressions on states can be.

Part III

Discussion

Chapter 7

Evaluation

In this chapter the design and implementation of OOPP and its QoS management features will be evaluated. First a closer evaluation of the performance cost of the extra flexibility introduced by reflection in OOPP will be done. Then two examples from the literature will be implemented using OOPP and some examples of the usage of OOPP in related research projects will be presented. Finally, a broader reflection on the current approach will summarise this chapter.

7.1 Flexibility and performance

As seen in chapter 5, flexibility has a higher priority than performance in this project. However, a closer look at some of the performance issues related to the implementation introducing this flexibility is necessary.

7.1.1 Local bindings and components

Often, flexibility is introduced by adding another level of indirection. Such an approach has the cost of the extra code that has to be executed during this indirection. An example is a method call over a local binding in OOPP (Figure 5.1 at page 58 illustrates the indirection introduced by a local binding).

When calling an empty method f of object a directly in Python the test computer[1] performs approximately 193.500 method calls per second. If the same method is called over a local binding approximately 18.300 method calls are executed per second. The extra levels of indirection introduced by the local binding have an extra cost that can not be ignored. However, most method calls are not done on empty methods. The cost of a method actually containing some code can differ a lot from the cost of the empty method f above. In such cases the extra cost introduced by the local binding becomes less significant. In the test setup each direct method call to f is measured to approximately 5μs.[2] The extra cost introduced by the local binding is measured to approximately 50μs. The extra cost introduced by the local binding becomes less significant when the added 50μs becomes less significant compared to the actual cost of the method call. A closer analysis of this extra cost shows that the two extra levels of indirection introduced by the interface pair of a local binding in fact introduces several extra name-lookups in Python. The example from Figure 5.1 on page 58 can be used to illustrate this. First Python has to locate interface reference j and the forwarding method inside j.[3] Then it has to locate the interface object o_i and the forwarding method inside o_i. Finally, the method f inside object a is located. In the Python language each of these steps involves a lookup in a hash-table (a Python dictionary). It is obviously room for improvements in this part of the OOPP implementation.

What happens when objects are wrapped into components? Exactly the same. The interfaces provided by a component is a direct mapping to (or naming of) the interfaces connected to the object. And since a local binding is just a relation (connection) between two interfaces, the component structure itself does not add any extra overhead to the method calls. This is also true for external interfaces of a composite component. However, extra costs of method calls through local bindings internally in the composite component have to be added. The edges of a component graph describing and connecting the contained components of a composite component are local bindings between interfaces of the contained components. Each edge introduces the extra cost of a local binding.

To summarise, local bindings introduce a significant extra cost in method calls. This extra cost becomes less significant when the cost of the called method increases.

7.1.2 Meta-objects

OOPP implements two different types of meta-objects. An encapsulation meta-object is used to access the encapsulation meta-model of an object, an interface, or a component. When creating an encapsulation meta-object of an object the contents of the object is copied to a new object instance an the original object instance becomes a shadow object that forwards method-calls and attribute access to the new copy (see Figure 5.7 on page 76). The consequence (and the reason for this) is that after the creation of the encapsulation meta-object every existing reference to the object are referring to the shadow object. The shadow object is used to control (and possibly manipulate) all access to the object. This extra level of introduction obviously introduce an extra cost in all access to the object. Table 7.1 lists the measured extra cost introduced by the encapsulation meta-object of object a when calling empty method f, reading integer attribute x, and updating integer attribute x. Method f and attribute x are members of object a.

As seen in Table 7.1, the extra cost introduced by the encapsulation meta-object is similar for method calls and attribute access. For method calls the significance of this extra cost depends a lot on the cost of the method itself (see Section 7.1.1 above). Access to attributes are not dependent on the implementation of the object (the class of the object)

Type of access	Num/second	Each call
Method call without meta-object	195K	5.1µs
Method call with meta-object	24K	41.3µs
Read without meta-object	380K	2.6µs
Read with meta-object	23K	42.6µs
Update without meta-object	410K	2.4µs
Update with meta-object	30K	33.0µs

Table 7.1 Measured extra cost introduced by the meta-object of an object.

and the extra cost accessing them will not be hidden in same way as the extra cost for accessing methods might be hidden. However, an object-oriented or a component-based programming model do not promote the possibility to access attributes directly.

A more efficient implementation of the encapsulation meta-object for objects should be possible (maybe with the cost of less flexibility). One possible implementation considered is to remove the shadow object and access the implementation of the actual object directly.

No extra cost is added when encapsulation meta-objects for interfaces and components are created. The reason is that this does not involve any changes to the implementation of the interfaces and the components itself. This is also true when creating a composition meta-object of a (composite) component.

To summarise, when the encapsulation meta-object of an object is created a significant extra cost is added to all access of the object. The significance of this extra cost depends a lot on the object classes (implementation) and the usage of the object.

7.2 Examples

The examples below illustrate the usage of OOPP and its features. Two examples from the literature demonstrate how OOPP can be used to solve the presented problems. The main motivation for these two examples is to demonstrate that the QoS management functionality of OOPP is flexible and powerful enough to implement such examples. Finally some examples of the usage of OOPP in related research projects are presented. The motivation for these examples is to show that OOPP is a suitable platform for continuing research on reflective middleware and QoS management.

7.2.1 Example I

In [56] Hafid and Bochmann present a quality of service adaptation example. Their example is a news-on-demand application where multimedia documents are stored are stored on different nodes. A system configuration consists of a client node (machine), a server

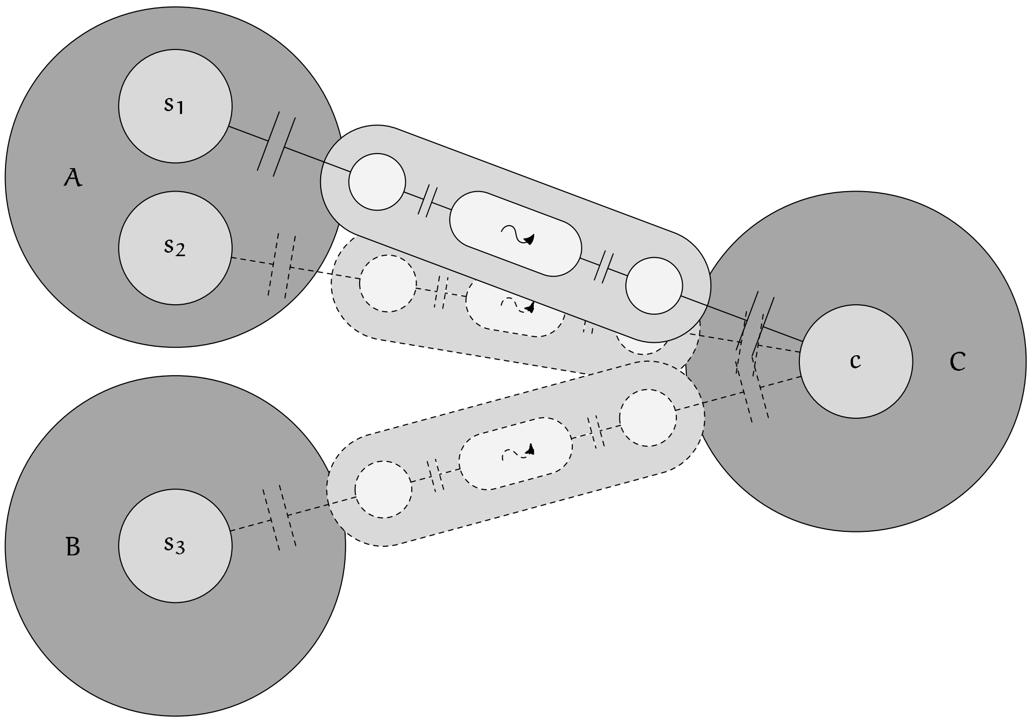


Figure 7.1 Example I with server capsule A providing service s_1 and s_2, server capsule B providing service s_3, and client capsule C containing client c. Only one server is connected to the client with the stream binding at a given time.

node, and a the network connecting the two nodes. A QoS manager maintains a configuration that allows the system to deliver the requested document with a presentation quality corresponding to the demands of the user. The provided service is video. The example include two capsules (servers) with different versions of the provided video (see Figure 7.1). Capsule A provides version 1 and 2 (s_1 and s_2) of the video, and Capsule B provides version 3 (s_3) of the video. Capsule A and B are typically located at different hosts with different network connections to the client c in capsule C. Version 1 and 3 of the video is of high quality (colour) and version 2 is of lower quality (grey-scaled).

The example starts with streaming version 1 of the video from capsule A to capsule C using the s_1–c configuration. If the s_1–c connection experiences congestion (signal c in Figure 7.2) the strategy selector switches to the s_3–c configuration (sends the signal s3 on the edge S_0–S_1 in Figure 7.2). If the s_3–c configuration is not able to support the delivery of the requested document the strategy selector switches to the s_2–c configuration (sends the signal s2 on the edge S_2–S_3 in Figure 7.2). If the system still is unable to support the delivery of the requested document the strategy selector asks the application to fail (sends the signal f on the edge S_4–S_5 in Figure 7.2). The state S_1 and S_3 of the automaton in Figure 7.2 are used to wait for the re-configured system to stabilise. The time T should be selected based on assumptions and experiences on how long the system needs to stabilise

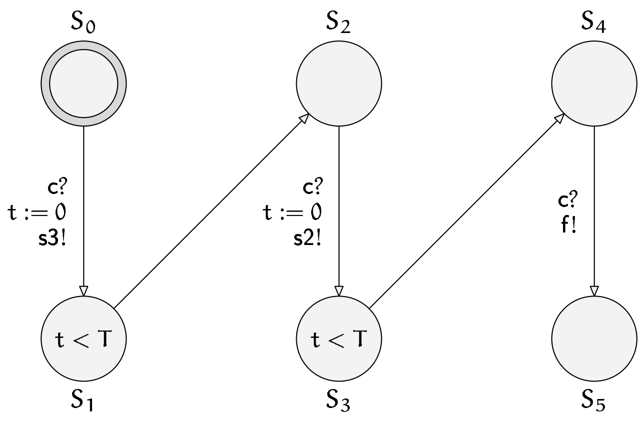


Figure 7.2 An automaton describing the behaviour of the strategy selector in Example I.

after a re-configuration. If T is to small the monitored values will monitor an non-stabilised system. The result could be that the management systems get garbage input and makes the wrong decisions. If T is to large the management decision process could be inactive too long not reacting on new problems that emerge.

The signal s3 and s2 are received by the strategy activators a_3 and a_2, respectively. Listing 7.1 includes the essential parts of the implementation of strategy activator a_3. The implementation of a_2 is similar.

In line 2 in Listing 7.1 an instance of the video service (implemented with class S) version 3 is created in capsule B. In line 5 a stream binding between capsule B and C is created. The interface references contain the information needed to locate their capsules (the standard stream and signal binding classes in the current implementation of OOPP accept both global interface references and capsule proxies as arguments to their constructors). In line 8 the binding is connected to the service s_3 (source) in capsule B. In line 11 the old stream binding is disconnected from the client in capsule C. We have delayed this step as late as possible to make the replacement as smooth as possible. In line 14 the new stream binding is connected to the client in capsule C. Finally in line 17 the old stream binding is deleted. Be aware that the calls to the getIRef method of capsule A in line 11 and capsule B in line 14 (inside the breakBinding and localBind calls respectively) are actually returning interface references from capsule C (and not from capsule A and B) since the stream binding components s1b and s3b are distributed composite components.

The description above of the OOPP implementation of the QoS adaptation example from [56] has illustrated how easy it is to implement such systems in OOPP. The reason is the expressive programming model, the reflective features, and the flexible QoS management scheme of OOPP. In this example the reflective features of OOPP can be used to monitor (inspect) the running system. The strategy selector is implemented as a simple OOPP automaton. This automaton could even be made simpler and more general if signals in OOPP are extended with arguments.[4] The strategy activators use standard capsule services

```
# Create service s3
s3 = B.mkComponent(S, ("Version 3",))

# Create binding between s3 and c
s3b = B.mkComponent(StreamBinding, (B.getIRef(s3,"src"),C.getIRef(c,"sink")))

# Connect binding at service side (source)
localBind(B.getIRef(s3,"src"), B.getIRef(s3b,"src"))

# Disconnect old binding at client side (sink)
breakBinding(A.getIRef(s1b,"sink"), C.getIRef(c,"sink"))

# Connect binding at client side (sink)
localBind(B.getIRef(s3b,"sink"), C.getIRef(c,"sink"))

# Remove old binding
A.delComponent(s1b)
```

Listing 7.1 Strategy activator a_3 from Example I.

to actually perform the adaptation. It is difficult to compare the OOPP implementation with the implementation presented in [56] since this article does not include sufficient implementation details.

7.2.2 Example II

The following example by Biersack and Geyer is originally presented in [17]. They present three different models for their synchronisation protocol (QoS management functions). Model 1 solves the problem of different but fixed delays on the network connections. Model 2 smoothes out jitter by using elastic buffers. Model 3 uses a feedback loop to re-synchronise the stream to handle the dynamic behaviour from the system (clock drift, changing network conditions and server drop outs). In the following the focus will be on model 3 since the QoS management work in OOPP only considers dynamic QoS management functions.

In model 3 a feedback filter and a control function is introduced. The functionality of these components are similar the functionality of the monitors and strategy selectors in OOPP. The goal of the feedback filter is to distinguish jitter from long-term disturbances. This is important since model 3 is used to handle long-term disturbances. The control function selects (and performs) the appropriate actions based on the input (signals) from the feedback filter.

Figure 7.3 illustrates this setup. The monitor m (the feedback filter) filters out jitter and can produce l and u signals. The l signal is generated when the lower watermark of the

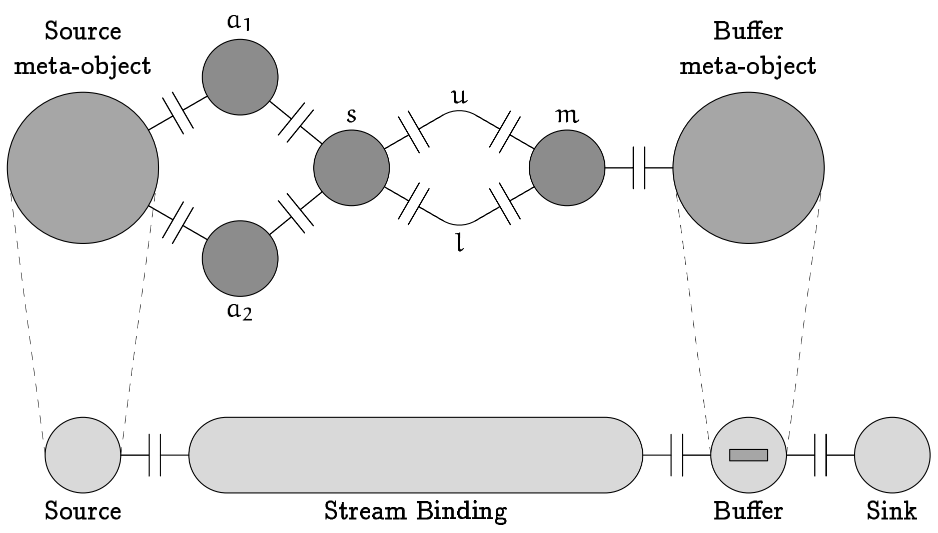


Figure 7.3 The Example II setup with the feedback filter (monitor m) and the control function (strategy selector s and strategy activators a_1 and a_2).

stream buffer is reached. The u signal is generated when the upper watermark of the stream buffer is reached. The l and u signals are received by the strategy selector s. The strategy selector s can either activate strategy activator a_1 or strategy activator a_2 by sending signals.

The strategy activators perform their tasks by accessing the meta-space of the source. The meta-space is accessed through a meta-object of the source. The monitor gets it input from the buffer component. This input can be generated by accessing the meta-space of the buffer. In this case, the encapsulation meta-object of the buffer component is used to monitor the buffer levels. A pre-method is added to a method that is called for every incoming media units. This pre-method checks the buffer level and generates a signal if lower or upper watermark is reached.

When the (smoothed) buffer level falls below the lower watermark the monitor m generates a l signal. When the (smoothed) buffer level exceeds the upper watermark the monitor m generates a u signal. Dependent on the history and time of its input the strategy selector s either picks a resynchronisation or an adaptation phase. In the resynchronisation phase a given number of media units is skipped (lower watermark) or the period of a given number of media units is paused (upper watermark). The offset (number of media units) to skip or to pause is either fixed or variable.

The adaptation phase is not described in details in this paper by Biersack and Geyer [17]. A description on how to choose between these two phases is also lacking. However, the adaptation phase and the selection between the resynchronisation and the adaptation phase should be easy to implement in OOPP. The meta-models of OOPP are made to inspect and adapt running software components. These features of OOPP matches the adaptation phase very well. Timed automata are a good tool to make decisions based on events, time and their history. And the paper indicates that the input to this selection process is (smoothed) lower and upper watermark events and their history. Since the strategy selectors (and monitors) in OOPP are implemented as timed automata they match the task to select between the resynchronisation and adaptation phase very well.

As seen above, the QoS management and reflective features of OOPP matches the need for QoS management functions described in [17] very well. However, one minor problem implementing the described setup was discovered. If variable offsets are used in the resynchronisation phase the strategy selector might become a little bit complex. The reason is that the signals sent to and received by the automata in OOPP do not include any arguments. The consequence is that there is no easy way to transmit the variable offset to the strategy activator. One solution is to create a new signal (and signal interface) for each possible value of the offset. This would make the strategy selector a little bit more complex than necessary. Another solution is to fix this problem in OOPP. Future versions of OOPP could be extended to allow parameters in signals (including in the input and output signals of automata components).

7.2.3 Other examples

OOPP has been used or has influenced several other research projects. All projects presented here are included in the Open-ORB or the V-QoS activities at Lancaster University.

Since OOPP is a prototype of the Open-ORB platform it has of course influenced the design of the current version of the Open-ORB platform [27].

The OOPP platform and its timed automata based QoS management functions have been used in the V-QoS project and the Open-ORB project to connect the world of formal specifications with a real running implementation of the system [22, 21]. This includes work on the role of reflection in QoS management functions [20, 21], specification of management components [30, 31], and formal support for dynamic QoS management [23].

Davis Sánches Gancedo demonstrated with the prototype QoSMonAuTA how OOPP could be used to dynamically adapt an audio stream delivered over a network with variable QoS. Network with variable QoS was simulated with tools implemented for this setup. The QoS management functions was modelled and implemented using the Timed Automata components of OOPP. In [53] he concluded that this approach could be put into practice and that his prototype application (using OOPP) successfully adapted to the simulated changes of network QoS.

Fábio Moreira Costa and his Meta-ORB prototype has been influenced by and reused some of the basic programming constructs of OOPP [41, 40]. The Meta-ORB project combines the use of meta-information management techniques with object-oriented reflection. The meta-information management techniques are used for the definition of (customised) platform configurations. Object-oriented reflection makes dynamic adaptation of running systems possible. A well defined meta-model integrates these two techniques and facilitates a consistent view of the different phases of the life-cycle of the platform (the same meta-information is used during design, development and at run-time).

Hector Duran Limon has designed and implemented a resource framework for adaptive (reflective) middleware [44, 43]. This resource framework is integrated in a modified and extended middleware platform influenced by and based on OOPP.

Katia Barbosa Saikoski has been working on adaptive groups in the Open-ORB platform [99]. Her implementation has been done using the OOPP platform (with some minor modifications).

7.3 Comments

OOPP provides an expressive programming model based on components and bindings with supporting capsules. Bindings and interfaces supporting signals and streams are included as basic programming structures in the platform. Other platforms like CORBA, Java RMI and DCOM/.NET provide none or only limited support for such programming structures. The consequence is that the programmer has to implement (and reinvent) such components based on lower level protocols not meant for application programmers using such platforms. A typical example is CORBA and streams. CORBA [89] has a limited support for the signalling protocol used for streams. The actual stream binding has to be implemented outside CORBA.

The provided services including constructors and factories are not overloaded with arguments to specify every possible configuration. The reason is that the reflective features of OOPP can be used to change the configuration. The result of this separation of concern is cleaner, more readable and more maintainable code.

The OOPP platform has been designed and implemented with a rich set of structural reflective features. The implementation has shown that some of these features have a higher cost than other features. The encapsulation meta-models of objects introduce (when used) some extra levels of indirection that might influence the running system. The encapsulation meta-models of interfaces and components, and the composition meta-models of components, do not add such an extra cost. The reasons are that the encapsulation meta-model of objects reifies the standard objects of the OOPP host language (Python) and the interfaces and components are provided by OOPP. The standard object model provided by Python is of course not designed to support the implementation of an encapsulation meta-model and to make it work in the current implementation an extra level of indirection (and the cost that follows it) is introduced. Interfaces and components of OOPP are designed and implemented carefully to support the reflective features of OOPP without adding any extra cost.

The provided structural reflective features fulfil the needs to inspect and adapt the structure of a running system. All the structural properties of the programming structures (objects, interfaces, components) can be inspected and modified. The usage of OOPP in real-world examples and in continuing research might conclude that the complexity and cost added by this rich set of features might be to high compared with how often they are used and how often they are necessary. The result could be a simpler model for reflection. In [27] a smaller set of structural reflective features are described.

The principle of reflection in OOPP makes its adaptive features flexible and expressive. This limits the constraints on how the platform and it components can adapt. It is not obvious in the two examples from the literature [56, 17] presented above that these systems can adapt in a completely different way when new demands are made upon them.

The timed automata based QoS management functions in OOPP use signals (and signal bindings) for the transportation of management information (events). Signal interfaces are the source and sink of this information. The current implementation of OOPP use only the signal itself to provide this information. The result is that every type of information has to be represented by its own signal and its own signal interface pair and signal binding. Example II above hinted that the possibility to provide arguments (information) with signals could have simplified the implementation of such a management setup. This extension should be considered in the future work on OOPP.

Notes

1. An IBM ThinkPad 570 366 MHz Pentium II with 128 Mbyte of memory, NetBSD 1.5.1, and Python 2.1.

2. This and all the other measurements below are done by repeating the operation in a loop with 10.000 iterations. The measured values include the execution of the loop itself.

3. It is even more complicated than this. Inside each forwarding methods Python has to locate a special method called __call__ and the attribute specifying the method to be called.

4. The automaton in Figure 7.1 could be implemented with only two states if the generated signals were allowed to pass arguments. The edge from S_1–S_2 could instead end up in state S_0. The edge from S_0–S_1 would then generate the signal s with an argument indicating what service the strategy activator should activate or if it should fail.

Chapter 8

Conclusion

This chapter concludes the work on OOPP presented in the previous chapters. First, a short summary of the work is given. Then, the major contributions are presented. Some ideas on future work are presented and finally some concluding remarks are given.

8.1 Summary

The OOPP platform presented is this work is a reflective middleware platform with an expressive programming model. The selected programming model is influenced by the RM-ODP standard and includes objects[1] and components with interfaces, capsules, and a naming service. Two interfaces can be connected with a binding. A binding is either a local binding or a binding component. Different types of interfaces and bindings are provided for operational methods, streams (continuous media), and signals. Components exists in capsules and are the building blocks of OOPP applications and the middleware itself. A composite component is a component containing or constructed from other components. The contained components of the composite component are included in the component graph of the composite component. The nodes of the graph are the contained components. The edges of the graph are local bindings. A composite component can be distributed. A typical distributed composite component is a binding with stub and skeleton components in different capsules (address spaces on different hosts).

Components are connected through their interfaces with bindings. These bindings define all interactions between components. A local binding is a direct mapping (relation) between two interfaces in the same capsule (address space). Binding components are used to connect interfaces of components in different capsules. A capsule is a managed address space providing a set of services to its local components and sometimes to remote components.

Different factories are also provided. They are usually used to create (and install) complex composite components. Without factories you create a set of components one-by-one and then glue them all together. A factory can create and install a large set of components in one operation. A factory can also include the process of establishing QoS management functions (see below).

Reflection is provided through different meta-models. These models are accessed using different types of meta-objects. OOPP implements structural reflection through the encapsulation and composite meta-models. Different versions of the encapsulation meta-objects can be used to inspect and manipulate objects, interfaces and components. The composite meta-objects are used to inspect and modify the component graph of composite components.

Dynamic QoS management is introduced using management components with three different roles. A monitor monitors (observes) the system and reports (using signals) about significant changes of the system behaviour. A strategy selector decides if the changing behaviour requires any adaptation of the system and selects an adaptation strategy. A strategy activator performs one adaptation strategy (selected by the strategy selector). Each possible adaptation strategy is represented by a strategy activator. Monitors and strategy selectors are (usually) implemented in OOPP using timed automata components. The behaviour of such a component is specified in a formal described timed automaton. A monitor component can use the reflective features of OOPP to inspect the behaviour of the system. A strategy activator can use the reflective features of OOPP to modify (adapt) the system. It is also possible to manage a QoS management setup using another level of management functions. This second level of management functions can be used to observe how well the first level management setup behave, and possibly replace it with a new one if it does not perform very well.

In the previous Chapters the usage of OOPP have been demonstrated, and Chapter 7 evaluated different aspects of the OOPP platform.

8.2 Major contributions

A prototype of the middleware platform described above has been implemented. This implementation has demonstrated that it is possible to implement a reflective middleware platform with an easy-to-use expressive programming model. This implemented programming model is powerful and flexible enough for a wide range of application domains. The programming model supports difficult programming constructs including continuous media (streams) and events (signals). The flexibility needed to support applications with changing demands and environment is provided through the concept of reflection. Reflection is introduced through a meta-object-protocol (MOP) implemented as meta-objects. The result is a clear distinction between the base programming model and the reflective features of OOPP. The flexibility added using reflection does not add any complexity to the base programming model.

The usage of the platform in several examples and related projects have shown that the provided programming environment matches the need of the programmer of complex distributed applications including continuous media an the need for great flexibility and adaptation.

The introduced QoS management functions in OOPP has proven to be flexible enough to support different application domains with a wide range of different management needs. The examples presented have shown that these QoS management functions fits well for typical difficult management scenarios from the related research literature.

The reflective features of OOPP have made it possible to implement the management tasks without introducing management-aware application and system components. The flexibility of reflection fits well with the goal of providing support for different types of application domains.

The prototype has also demonstrated that it is possible to base these running management functions on formal description techniques. Timed automata describing the behaviour of management functions is implemented as management components doing the (formally) described management task at run-time. These components consumes and produces signals used to monitored behaviour and selected management strategies.

8.3 Future work

The future work on OOPP includes extension of the programming model, the reflective features, and the QoS management functionality.

The programming models should be extended with a set of high-level binding components supporting the need of different applications. These includes more complex stream bindings with buffers and real-time characteristics, bindings for asynchronous interaction between components (messaging service), and bindings supporting group communication (see [99] for work on groups and group communication in Open-ORB). Experiences with signals and signals bindings have shown that the possibility to pass arguments with signals would be useful in several application types. Future work on OOPP should include signals with arguments and an implementation of automata components that can receive and send such signals. Security is another aspect that has been completely ignored in the current design and implementation of OOPP. Future work should look into this very difficult topic. Secure access to components and capsules in OOPP should be as flexible as the rest of the OOPP programming model.

The future work on reflective features in OOPP will focus on behavioural reflection. Behavioural reflection (described in Chapter 3) is in Open-ORB represented by the environment and the resource meta-model. The resource meta-model could be based on the work of Hector Duran Limon [44, 43]. The environment meta-model might be inspired by the concurrent programming model of ATOM [91]. The ATOM project does not include reflection, but separation of concern is a strong motivation. Concurrent behaviour is separated from the functional implementation of objects. Another research topic is how security mechanisms is accessible (and made flexible) through the provided meta-models.

QoS management in current OOPP is only concerned with dynamic management functions. Future work should include a complete QoS management model in an OOPP platform including both structural and behavioural reflection. Some important contribution to such a task can be found in the work on the Meta-ORB of Fábio Moreira Costa [41, 40].

8.4 Concluding remarks

The reason to introduce reflection is to make a system more flexible. Traditional flexible systems becomes complex system both from the users view-point and from the system programmers view-point. The reason is that such an system must include all the functionality needed to make it flexible. All the different behaviours of the system has to be implemented and installed in the system before it is used. A reflective system makes it possible to observe and change the behaviour of the system dynamically by accessing the meta-object-protocol (MOP) of the system. And since this is done through a MOP the complexity of the base programming model is not increased. This separation of concern is essential in all reflective systems. For the same reason is the MOP itself often separated in different meta-models concerning different aspects of the behaviour of the system.

An application programmer using a reflective middleware should use the same concept of separation of concern. While implementing the different components and their interaction she should focus on the functional aspects. The non-functional behaviour should be manipulated through the different meta-models provided by the reflective middleware platform.

QoS management is a difficult task. The possibility to formally reason about the behaviour of a system and its QoS management setup is an important tool when developing such systems. A formally described management setup can also be used in simulations and analyses of the management behaviour. Since OOPP can run such formally described management component directly it can easily be used together with such simulation and analyse tools.

The goal of the OOPP project is to make the development of complex distributed application less complicated. The expressive programming model, its reflective features and the QoS management functions based on formally described management behaviour is the contribution from the OOPP project to reach this goal.

Notes

1. Components wrap the actual objects in the OOPP programming model. Objects contain the actual implementation (and state) of components.

— ∗ —

Part IV

Appendix

Appendix A

Author and editor index

Appendix B

Bibliography

[1] Jan Øyvind Aagedal, *Quality of service support in development of distributed systems*, Ph.D. thesis, Department for Informatics, University of Oslo, June 2001. {23}

[2] Gul Agha, *The structure and semantics of actor languages*, The Proceedings of REX School/Workshop Foundations of Object-Oriented Languages (Noordwijkerhout, The Netherlands) (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), Lecture Notes in Computer Science, vol. 489, Springer-Verlag, May/June 1990, pp. 1–59. {18}

[3] Rajeev Alur and David L. Dill, *The theory of timed automata*, Proceedings of the REX Workshop 1991, Real-Time: Theory in Practice (Mook, The Netherlands) (J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, eds.), Lecture Notes in Computer Science, vol. 600, Springer-Verlag, June 1991, pp. 45–73. {88}

[4] ———, *A theory of timed automata*, Theoretical Computer Science **126** (1994), 183–235. {88}

[5] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi, *Communication modeling by channel reification*, Workshop on Advances in Languages for User Modeling, in Sixth International Conference on User Modeling (Chia Laguna, Sardinia Italia), June 1997, pp. 1–9. {21}

[6] Anders Andersen, *A note on reflection in Python 1.5*, Distributed Multimedia Research Group Report MPG-98-05, Lancaster University, UK, March 1998. {12, 19}

[7] ———, *The Open-ORB Python prototype API*, NORUT IT Report IT302/2-99, NORUT IT, October 1999. {11}

[8] ______, *A reflective component-based middleware in Python*, The Eighth International Python Conference (Arlington, Virginia, USA), January 2000, (short talk). {11}

[9] Anders Andersen, Gordon S. Blair, and Frank Eliassen, *OOPP: A reflective component-based middleware*, NIK 2000 (Bodø, Norway), November 2000. {8, 11}

[10] ______, *A reflective component-based middleware with quality of service management*, PROMS 2000, Protocols for Multimedia Systems (Cracow, Poland), October 2000. {8, 9}

[11] Anders Andersen, Gordon S. Blair, Vera Goebel, Randi Karlsen, Tage Stabell-Kulø, and Weihai Yu, *Arctic Beans: Configurable and reconfigurable enterprise component architectures*, IEEE Distributed Systems Online **2** (2001), no. 7, Work in Progress session at Middleware 2001, Heidelberg, Germany. {10}

[12] ______, *Arctic Beans: Flexible and open enterprise component architectures*, NIK 2001 (Tromsø, Norway), November 2001. {10}

[13] Phillip G. Armour, *The case for a new business model*, Communications of the ACM **43** (2000), no. 8, 19–22. {8}

[14] ______, *The five orders of ignorance*, Communications of the ACM **43** (2000), no. 10, 17–20. {8}

[15] C. Aurrecoechea, A. T. Campbell, and L. Hauw, *A survey of QoS architectures*, ACM Multimedia Systems **6** (1998), no. 3, Lancaster University Report MPG-95-18. {23}

[16] Philip A. Bernstein, *Middleware: A model for distributed system services*, Communications of the ACM **39** (1996), no. 2, 86–98. {15, 25}

[17] Ernst Biersack and Werner Geyer, *Synchronized delivery and playout of distributed stored multimedia streams*, ACM Multimedia Systems **7** (1999), no. 1, 70–90. {108–110, 112}

[18] Andrew D. Birrell and Bruce Jay Nelson, *Implementing remote procedure call*, ACM Transactions on Computer Systems **2** (1984), no. 1, 39–59. {3, 15}

[19] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, *Distribution and abstract types in Emerald*, IEEE Transactions on Software Engineering **SE-13** (1987), no. 1, 65–76. {53}

[20] Gordon S. Blair, Anders Andersen, Lynne Blair, and Geoff Coulson, *The role of reflection in supporting dynamic QoS management functions*, Seventh International Workshop on Quality of Service (IWQoS '99) (London, UK), Distributed Multimedia Research Group Report, no. MPG-99-03, IEEE/IFIP, Lancaster University, June 1999. {11, 110}

[21] Gordon S. Blair, Anders Andersen, Lynne Blair, Geoff Coulson, and David Sánchez Gancedo, *Supporting dynamic QoS management functions in a reflective middleware platform*, IEE Proceedings – Software **147** (2000), no. 1, 13–21. {8, 9, 110}

[22] Gordon S. Blair, Lynne Blair, Anders Andersen, and Trevor Jones, *A formal view of aspects in the development of component-based distributed systems*, SCI'2000 invited session on Generative and Component-Based Software Engineering (Orlando, Florida, USA), July 2000. {11, 110}

[23] ———, *Formal support for dynamic QoS management in the development of open component-based distributed systems*, IEE Proceedings – Software **148** (2001), no. 3, 83–92, Special issue on generative and component-based software engineering. {10, 110}

[24] Gordon S. Blair, Lynne Blair, Howard Bowman, and Amanda Chetwynd, *Formal specification of distributed multimedia systems*, UCL Press, 1998. {5}

[25] Gordon S. Blair, Fabio Costa, Geoff Coulson, Hector Duran, Nikos Parlavantzas, Fabien Delpiano, Bruno Dumant, Franćois Horn, and Jean-Bernard Stefani, *The design of a resource-aware reflective middleware architecture*, Proceeding of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99) (St-Malo, France), Lecture Notes in Computer Science, vol. 1616, Springer-Verlag, 1999, pp. 115–134. {33}

[26] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran, Nikos Parlavantzas, and Katia B. Saikoski, *A principled approach to supporting adaptation in distributed mobile environments*, 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000) (Limerick, Ireland), June 2000. {11, 79}

[27] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia B. Saikoski, *The design and implementation of Open ORB 2*, IEEE Distributed Systems Online **2** (2001), no. 6. {10, 110, 112}

[28] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas, *An architecture for next generation middleware*, Middleware'98, September 1998. {20, 25}

[29] Gordon S. Blair and Jean-Bernard Stefani, *Open distributed processing and multimedia*, Addison-Wesley, 1998. {27}

[30] Lynne Blair, Gordon S. Blair, and Anders Andersen, *Separating functional behaviour and performance constraints: Aspect-oriented specification*, Distributed Multimedia Research Group Report MPG-98-07, Computing Department, Lancaster University, May 1998. {12, 110}

[31] ———, *A multi-paradigm specification technique supporting the synthesis of qos management components*, Tech. report, Distributed Multimedia Research Group, Lancaster University, February 1999. {11, 110}

[32] Lynne Blair, Trevor Jones, and Gordon S. Blair, *Stochastically enhanced timed automata*, Proceedings of the Fourth International Workshop on Formal Methods for Open Object-Based Distributed System (FMOODS'99) (Standford, California), 2000. {88}

[33] Gregor V. Bochmann and Abdelhakim Hafid, *Some principles for quality of service management*, Distributed System Engineering Journal **4** (1997), 16–27. {23}

[34] Per Brinch Hansen, *Operating system principles*, Prentice Hall Series in Automatic Computation, Prentice Hall, 1973. {25}

[35] Walter Cazzola and Massimo Ancona, *mChaRM: a reflective middleware for communication-based reflection*, Report DISI-TR-00-09, DISI, Universitâ degli Studi di Genova, May 2000. {21}

[36] Martin Chapman and Stefano Montesi, *Overall concepts and principles of TINA*, TINA Baseline TB-MDC.018-1.0-94, TINA, February 1995. {3}

[37] Leonardo Chiariglione, *MPEG: A technological basis for multimedia applications*, IEEE Multimedia **2** (1995), no. 1, 85–89. {52}

[38] Shigeru Chiba and Takashi Masuda, *Designing an extensible distributed language with a meta-level architecture*, ECOOP '93 — Object Oriented Programming (Kaiserslautern, Germany) (Oscar M. Nierstrasz, ed.), Lecture Notes in Computer Science, vol. 707, Springer-Verlag, July 1993, pp. 482–501. {20}

[39] Pierre Cointe, *Reflective languages and metalevel architectures*, ACM Computing Surveys **28A** (1996), no. 4. {19}

[40] Fabio Costa, *Combining meta-information management and reflection in an architecture for configurable and reconfigurable middleware*, Ph.D. thesis, Computing Department, Lancaster University, August 2001. {110, 116}

[41] Fabio Costa and Gordon S. Blair, *The role of meta-information management in reflective middleware*, ECOOP'2000 Workshop on Reflection and Metalevel Architectures (Sophia Antipolis and Cannes, France), June 2000. {110, 116}

[42] Fabio Costa, Gordon S. Blair, and Geoff Coulson, *Experiments with reflective middleware*, ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems (Brussels, Belgium), Lecture Notes in Computer Science, vol. 1543, July 1998, pp. 390–391. {32}

[43] Hector Duran and Gordon S. Blair, *Configuring and reconfiguring resources in middleware*, 1st International Symposium on Advanced Distributed Systems (Guadalajara, Jalisco, Mexico), March 2000. {111, 115}

[44] ———, *A resource management framework for adaptive middleware*, 3th IEEE International Symposium on Object-oriented Real-time Distributed Computing (Newport Beach, California, USA), March 2000. {111, 115}

[45] Wayne W. Eckerson, *Three-tier client/server architecture*, Open Information Systems **10** (1995), no. 1, 3–22. {15, 25}

[46] Frank Eliassen, Anders Andersen, Gordon S. Blair, Fabio Costa, Geoff Coulson, Vera Goebel, Øyvind Hanssen, Tom Kristensen, Thomas Plageman, Hans Ole Rafaelsen, Katia B. Saikoski, and Weihai Yu, *Next generation middleware: Requirements, architecture, and prototypes*, 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99) (Tunisia, South Africa), December 1999. {11}

[47] Frank Eliassen, Tom Kristensen, Thomas Plageman, and Hans Ole Rafaelsen, *MULTE-ORB: Adaptive QoS aware binding*, Workshop on Reflective Middleware at Middleware 2000 (New York, USA), April 2000. {20}

[48] Frank Eliassen and John R. Nicol, *A flexible type checking model for stream interface binding*, Proceedings of the International Workshop on Multimedia Software Development (Berlin, Germany), IEEE Computer Society Press, March 1996, pp. 52–60. {53}

[49] Wolfgang Emmerich, *Engineering distributed objects*, John Wiley & Sons, 2000. {68}

[50] Kazi Farooqui, Luigi Logrippo, and Jan de Meer, *The ISO reference model for open distributed processing: an introduction*, Computer Networks and ISDN Systems **27** (1995), no. 8, 1215–1229. {55}

[51] Tom Fitzpatrick, Gordon S. Blair, Geoff Coulson, Nigel Davies, and Philippe Robin, *Supporting adaptive multimedia applications through open bindings*, Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98) (Annapolis, Maryland, US), May 1998. {32, 51}

[52] R. P. Gabriel, J. L. White, and Daniel G. Bobrow, *CLOS: Integrating object-oriented and functional programming*, Communications of the ACM **34** (1991), no. 9, 28–38. {19}

[53] David Sánches Gancedo, *QoSMonAuTA, QoS monitoring and adaptation using timed automata*, Master's thesis, Lancaster University, UK, September 1999. {110}

[54] Brendan Gowing and Vinny Cahill, *Meta-object protocols for C++: The Iguana approach*, Proceedings of the Reflection'96 Conference (Gregor Kiczales, ed.), April 1996, pp. 137–152. {20}

[55] Abdelhakim Hafid and Gregor V. Bochmann, *An approach to quality of service management for distributed multimedia applications*, Procceedings of the Third

IFIP International Conference on Open Distributed Computing (Brisbane, Australia), 1995. {23}

[56] ______, *Quality-of-service adaptation in distributed multimedia applications*, ACM Multimedia Systems **6** (1998), no. 5, 299–315. {105, 107, 108, 112}

[57] Øyvind Hanssen, *FlexiNet — binding framework*, Tech. Report APM.2057.02.00, ANSA Phase III, APM, October 1997. {20}

[58] Øyvind Hanssen and Frank Eliassen, *Towards a QoS aware binding model*, SYBEN'98 (Zürich), May 1998. {20}

[59] Richard Hayton, *FlexiNet open ORB framework*, Tech. Report APM.2047.01.00, ANSA Phase III, APM, 1997. {20}

[60] ISO/IEC, *Open distributed processing reference model, part 1: Overview*, ITU-T Rec. X.901 | ISO/IEC 10746-1, ISO/IEC, 1995. {3, 17, 28, 37, 55}

[61] ______, *Open distributed processing reference model, part 2: Foundations*, ITU-T Rec. X.902 | ISO/IEC 10746-2, ISO/IEC, 1995. {17, 28, 38}

[62] ______, *Open distributed processing reference model, part 3: Architecture*, ITU-T Rec. X.903 | ISO/IEC 10746-3, ISO/IEC, 1995. {17, 28, 29, 43, 44, 47}

[63] ______, *QoS – basic framework*, ISO Report ISO/IEC JTC1/SC21 N9309, ISO/IEC, 1995. {22}

[64] ITU, *Recommendation H.263 (02/98) – video coding for low bit rate communication*, ITU-T H.263, International Telecommunication Union, 1998. {52}

[65] Brad Curtis Johnson, *A distributed computing environment framework: An OSF perspective*, Distributed Open Systems in Perspective — Proceedings of the Spring 1991 EurOpen Conference (Tromsø, Norway), EurOpen, May 1991, pp. 69–87. {3}

[66] Randy H. Katz, *Adaptation and mobility in wireless information systems*, IEEE Personal Communications **1** (1994), no. 1, 6–17. {26}

[67] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, *The art of the metaobject protocol*, The MIT Press, 1991. {18, 19, 27}

[68] Steven S. King, *Middleware!*, Data Communications **21** (1992), no. 4, 58–67. {15, 25}

[69] Mary Kirtland, *The COM+ programming model makes it easy to write components in any language*, Tech. report, Microsoft Corporation, November 1997. {17}

[70] Fabio Kon, Manuel Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and Roy H. Campbell, *Monitoring, security and dynamic configuration with the dynamicTAO reflective ORB*, Proceedings of Middleware 2000 (New York, USA), April 2000. {20, 23}

[71] Baochun Li and Klara Nahrstedt, *A control-based middleware framework for quality of service adaptations*, ieee-jsac (1999). {23}

[72] ———, *Dynamic reconfiguration for complex multimedia applications*, Proceedings of IEEE International Conference on Multimedia Computing and Systems (Florence, Italy), June 1999. {23}

[73] Jesse Liberty, *Attributes and reflection*, Programming C#, A Nutshell Handbook, O'Reilly & Associates, Inc., July 2001. {17}

[74] T. D. C. Little and D. Venkatesh, *Client-server metadata management for the delivery of movies in a video-on-demand system*, Proceedings of the 1st International Workshop on Services in Distributed and Networked Environments, SDNE (Prague, Czech Republic), June 1994, pp. 11–18. {23}

[75] David C. Luckham, James Vera, and Sigurd Meldal, *Three concepts of system architecture*, Tech. Report CSL-TR-95-674, Computer Systems Laboratory, Stanford University, July 1995. {29}

[76] Mark Lutz, *Programming Python*, 2 ed., A Nutshell Handbook, O'Reilly & Associates, Inc., 2001. {8, 19, 55}

[77] E. Madelaine and R. de Simone, *FC2: Reference manual, version 1.1*, Tech. report, INRIA Sophia-Antipolis, 1993. {94}

[78] Pattie Maes, *Concepts and experiments in computational reflection*, Proceedings of OOPSLA '87, Sigplan Notices, vol. 22, ACM Press, October 1987, pp. 147–155. {19, 26}

[79] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa, *Hybrid group reflection architecture for object-oriented concurrent reflective programming*, Proceedings of the European Conference on Object-Oriented Programming, ECOOP '91 (Geneva, Switzerland) (Pierre America, ed.), Lecture Notes in Computer Science, vol. 512, Springer-Verlag, July 1991. {27}

[80] Peter M. Maurer, *Components: What if they gave a revolution and nobody came?*, Computer **33** (2000), no. 6, 28–35. {3}

[81] J. McAffer, *Engineering the meta-level*, Proceedings of the Reflection'96 Conference (Gregor Kiczales, ed.), April 1996, pp. 39–61. {20}

[82] ———, *Meta-level architecture support for distributed objects*, Proceedings of Reflection'96 (San Francisco) (Gregor Kiczales, ed.), 1996, pp. 39–62. {18, 20, 33}

[83] Glen McCluskey, *Remote method invocation: Creating distributed java-to-java applications*, Tech. report, Sun Microsystems, Inc., October 1997. {3}

[84] Microsoft, *DCOM technical overview*, Microsoft Windows NT Server white paper, Microsoft Corporation, 1996. {3, 17}

[85] Per Harald Myrvang, Tage Stabell-Kulø, and Anders Andersen, *Flexible authentication and delegation for distributed components*, Tech. report, Department of Computer Science, University of Tromsø, 2001, (submitted Middleware 2001). {10}

[86] Klara Nahrstedt and Jonathan M. Smith, *The QOS broker*, IEEE Multimedia **2** (1995), no. 1, 53–67. {23}

[87] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola, *Object orientation in hetrogeneous distributed computing system*, Computer **26** (1993), no. 6, 57–67. {27}

[88] Object Managment Group, *The common object request broker: Architecture and specification*, Tech. Report 96.3.4, Object Managment Group, July 1995, (revision 2.0). {3, 16}

[89] ———, *The common object request broker: Architecture and specification*, Tech. report, Object Managment Group, February 2001, (revision 2.4.2). {7, 16, 111}

[90] H. Okamura, Y. Ishikawa, and M. Tokoro, *AL-1/D: A distributed programming system with multi-model reflection framework*, Proceedings of the Workshop on New Models for Software Architecture, November 1992. {19, 20, 27, 31}

[91] Michael Papathomas and Anders Andersen, *Concurrent object-orieted programming in Python with ATOM*, The Sixth International Python Conference (San Jose, California), October 1997, pp. 77–87. {12, 32, 115}

[92] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek, *A resource allocation model for QoS management*, Proceedings of the IEEE Real-Time Systems Symposium, December 1997. {23}

[93] Kerry Raymond, *Reference model of open distributed processing (RM-ODP): Introduction*, Tech. report, CRC for Distributed System Technology, Centre for Information Technology Research, University of Queensland, 1993. {55}

[94] ReTINA, *Extended DPE resource control framework specifications*, ReTINA Deliverable AC048/D1.01xtn, France Telecom – CNET and GMD Fokus, January 1999, ACTS Project AC048, An Industrial-Quality TINA-Compliant Real-Time DPE. {33}

[95] Manuel Roman, Fabio Kon, and Roy H. Campbell, *Design and implementation of runtime reflection in communication middleware: the dynamicTAO case*, ICDCS'99 Workshop on Middleware (Austin, Texas), May 1999. {20}

[96] ______, *LegORB and ubiquitous CORBA*, Workshop on Reflective Middleware at Middleware 2000 (New York, USA), April 2000. {20}

[97] Manuel Roman, Ashish Singhai, Dulcineia Carvalho, Christopher Hess, and Roy H. Campbell, *Integrating PDAs into distributed systems: 2k and PalmORB*, International Symposium on Handheld and Ubiquitous Computing (HUC 99) (Karlsruhe, Germany), September 1999. {20}

[98] William Ruh, Thomas Herron, and Paul Klinker, *IIOP complete: Understanding CORBA and middleware interoperability*, Object Technology Series, Addison-Wesley, 2000. {7}

[99] Katia B. Saikoski and Geoff Coulson, *Adaptive groups in Open ORB*, Proceedigns of the CAiSE '99 Doctoral Consortium (Heidelberg, Germany), June 1999. {111, 115}

[100] Ashish Singhai, Aamod Sane, and Roy H. Campbell, *Quarterware for middleware*, 18th IEEE International Conference on Distributed Computing Systems (Amsterdam, The Netherlands), May 1998. {21}

[101] Brian Cantwell Smith, *Procedural reflection in programming languages*, Ph.D. thesis, Massachusetts Institute of Technology, 1982. {18, 19}

[102] Sun Microsystems, *Java core reflection*, Tech. report, Sun Microsystems, Inc., 1998. {19}

[103] Clemens Szyperski, *Component software, beyond object-oriented programming*, ACM Press/Addison-Wesley, 1998. {4, 29}

[104] Thuan Thai and Hoang Q. Lam, *.NET framework*, A Nutshell Handbook, O'Reilly & Associates, Inc., 2001. {3}

[105] Rob van der Linden, *An overview of ANSA*, Architecture Report APM.1000.01, ANSA, Architecture Projects Managment Limited, July 1993. {3}

[106] Guido van Rossum, *Glue it all together with Python*, Workshop on Compositional Software Architectures (Monterey, California), January 1998, (position paper). {55}

[107] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran, *Adaptive and reflective middleware for QoS-enabled CCM applications*, IEEE Distributed Systems Online **2** (2001), no. 5. {23}

[108] Takuo Watanabe and Akinori Yonezawa, *Reflection in an object-oriented concurrent language*, OOPSLA '88 Proceeding, Sigplan Notices, vol. 28, ACM Press, September 1988, pp. 306–315. {18, 19, 27, 28, 31, 33}

[109] M. Wegdam, D.-J. Plas, A. van Halteren, and B. Nieuwenhuis, *Using reflection in a managemenr architecture for CORBA*, Proceedings of the 11th IFIP/IEEE

International Workshop on Distributed Systems, Operations & Management (DSOM 2000) (Austin, Texas, USA), December 2000. {23}

[110] Sara Williams and Charlie Kindel, *The component object model: A technical overview*, Dr. Dobb's Journal (1994). {16}

[111] Zhixue Wu, *Design and implementation of a transaction service for Java*, Technical Report APM.1923.00.01, ANSA Phase III, APM, January 1997. {20}

[112] Zhixue Wu and Scarlet Schwiderski, *Reflective Java: Making Java even more flexible*, External Paper APM.1936.02, ANSA Phase III, APM, February 1997. {20}

[113] Y. Yokote, *The Apertos reflective operating system: The concept and its implementation*, Proceedingss of OOPSLA'92, Sigplan Notices, vol. 28, ACM Press, 1992, pp. 414–434. {18, 20}

Appendix C

Compendium

In «A Reflective Component-Based Middleware with Quality of Service Management» (se Section C.1 below) OOPP is presented as reflective middleware with the possibility to inspect, adapt and extend the components of the system to satisfy the requirements of a given application. Quality of service management is achieved with management components with different roles (monitors, strategy selectors and strategy activators). Timed automata is presented as one way of specifying the behaviour of such management components. The paper has been published in the Proceedings of the 5th International Conference on Protocols for Multimedia Systems (PROMS 2000, Cracow, Poland, October 2000).

In «Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform» (se Section C.2 below) the role of reflection in supporting the dynamic QoS management functions of monitoring and adaptation is considered. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies. The paper has been published in IEE Proceedings – Software (Volume 147, Issue 1, June 2000)[1].

C.1 Proceedings PROMS 2000

A REFLECTIVE COMPONENT-BASED MIDDLEWARE WITH QUALITY OF SERVICE MANAGEMENT

Anders Andersen[1] *Gordon S. Blair* [2,1] *Frank Eliassen*[3,1]
[1]Department of Computer Science, University of Tromsø,
N-9037 Tromsø, Norway, e-mail: aa@computer.org
[2]Computing Department, Lancaster University,
Lancaster LA1 4YR, UK, e-mail: gordon@comp.lancs.ac.uk
[3]Department of Informatics, University of Oslo,
N-0316 Oslo, Norway, e-mail: frank@ifi.uio.no

ABSTRACT

The role of middleware is to present a unified programming model to application writers and mask out the problems of heterogeneity and distribution. However, new application types need support for multimedia, real-time and mobility. This support should be configurable to satisfy the requirements from a wide variety of applications. But this is not enough. Mobility and continuous media requires the possibility to inspect and adapt the support offered at run-time. This can be done by adopting an open engineering approach for the design of the middleware platform. Reflection provides a principled (as opposed to ad hoc) means of achieving open engineering. The Open-ORB Python Prototype (OOPP) presented in this paper implements a reflective middleware with the possibility to inspect, adapt and extend the components of the system to satisfy the requirements of a given application.

1 INTRODUCTION

The role of middleware is to present a unified programming model to application writers and mask out the problems of heterogeneity and distribution [1, 2, 3]. However, the middleware has to remain responsive to challenges and demands from existing and new type of applications, including (i) support for multimedia, (ii) real-time requirements, and (iii) increasing mobility. Given this, it should be possible to configure the underlying support offered by the middleware platform to satisfy the requirements from a wide variety of applications. Example of such configurations are scheduling policies, special protocols for multimedia, and resource management.

Another important requirement is the possibility to inspect and adapt the support offered at run-time. This can be done by adopting an open engineering approach for the design of the middleware platform. Reflection provides a principled (as opposed to ad hoc) means of achieving such open engineering.

Reflection initially emerged in the programming language community as an important technique to introduce more flexibility and openness into the design of programming languages. This work tries to use these techniques to introduce flexibility and openness into the area of distributed systems, in general, and middleware, in particular.

The aim of this paper is to present a prototype implementation of such a middleware platform. The Open-ORB Python Prototype (OOPP) is based on an architecture under development in the Open-ORB project [4, 5]. The paper is structured as follows. Section 2 gives an introduction to the Open-ORB architecture. Section 3 presents the programming structures and section 4 presents the infrastructure of the prototype. Section 5 shows how reflection is provided through the implementation of meta-models. Section 6 discusses QoS management in the implemented prototype. Section 7 gives an overview over related work and section 8 presents some ideas for future work. Section 9 concludes the paper.

2 OPEN-ORB

2.1 Reflective middleware

The reflection hypothesis introduced by Smith in 1982 [6] states:

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operation and structures.*

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a meta-object protocol (MOP) which defines the services available at the meta-level. Initially, Smith's work catalysed a large body of work to the field of programming language design [7, 8, 9]. Later reflection has been applied to operating systems [10] and, more recently, distributed systems [11].

Current middleware platforms adopt a black box philosophy, whereby the implementation details are hidden from the application programmer. However, there is increasing evidence though that the black box philosophy is becoming untenable. For example, the OMG have recently added internal interfaces to CORBA to support services such as transactions and security, and the Portable Object Adapter is another CORBA attempt to introduce openness. All these approaches are rather ad hoc. The Open-ORB project tries to address this by providing openness in the middleware design in a principled way through the concept of reflection.

2.2 General principles

Open-ORB adopts a component-based model of computation. A component is a unit for composition and independent deployment [12]. This is important when possible replacements and restructuring of components in a system are determined. In Open-ORB components are used for the structuring of the middleware platform itself; deployable components are the foundation for flexibility (reconfiguration) of Open-ORB. The Open-ORB component model is derived from the computational viewpoint of RM-ODP [13] where (i) components have interfaces, (ii) interfaces for continuous media are supported and (iii) explicit bindings can be created between compatible interfaces.

In addition, Open-ORB meta-level supports per interface (or sometimes per component) meta-spaces. This is necessary in a heterogeneous environment where components will have varying capacities for reflection. Such a solution also provides a fine level of control over the support provided by the middleware platform; a corollary of this is that problems of maintaining integrity are minimised due to the limited scope of change. In situations where it is useful to access sets of meta-spaces in a single action the use of groups are possible [14].

Finally, Open-ORB structures the meta-space as a number of closely related but distinct meta-space models. This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [15]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects.

2.3 Meta space

In reflective systems, structural reflection is concerned with the content of a given component [8]. In Open-ORB, this aspect of meta-space is represented by two distinct meta-models, namely the encapsulation and composition meta-models. The encapsulation meta-model provides access to the representation of a particular interface or component in terms of its set of methods, attributes and other key properties (including its inheritance structure). This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. The composition meta-model provides access to the component graph of a component. The component graph of a composite component describes the contained components and how these components are related (connected). The component graph can be inspected and altered using the composition meta-model of an composite component. All interfaces of a component provides the same composition meta-model.

Behavioural reflection is concerned with activity in the underlying system [8]. This is represented by the environmental meta-model. In terms of middleware, the underlying activity equates to functions such as message arrival, enqueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [8, 11].

Multimedia, real-time and mobile applications need access to the resources and the resource management of the system. Open-ORB introduces the resource meta-model [16, 17] for this task. This model is concerned with the allocation and management of resources associated with any activity in the system.

2.4 OOPP

A prototype implementation of the suggested Open-ORB middleware architecture has been implemented in Python [18, 19]. The Open-ORB Python Prototype (OOPP) is influenced by the Open Distributed Processing Reference Model (RM-ODP) [20, 21]. In addition, it provides structural reflection through the encapsulation and composition meta-models. The purpose of OOPP has been to validate the Open-ORB architecture by illustrating the utility of its various adaptation mechanisms.

Python has been selected as implementation language for several reasons. The most important reason is the existing reflective features of the languages [22]. The structural meta-models have proven to be relatively easy to build on top of these reflective features. Another argument for choosing Python is its interpreted nature and flexible typing that makes it ideal for prototyping. The good support for distributed programming is also preferable.

3 PROGRAMMING STRUCTURES

3.1 Interfaces and local bindings

An interface of an object defines a subset of the interactions of that object [13]. A method call to a method of the object and a method call from the object (to a method of another object) are examples of such interactions. Different interface types for operational methods, signals and streams are available. The operational interface type is the basic interface type that all other interface types are based on. It provides a set of exported and imported methods and it is associated with a given object. An exported method of an object is a method of that object made available through an interface. An imported method of an object is an external method made available to the object through an interface.

A local binding is an establishing behaviour between two interfaces. A local binding connects the exported methods of one interface to the imported methods of another interface and vice versa. The two objects `a` and `b` in Figure 1 have an interfaces `i` and an interface `j`, respectively. Inter-

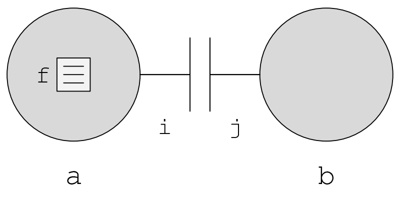


Figure 1. A local binding

face `i` and `j` are bound with a local binding. Object `b` can call method `f` of object `a` through its interface `j`.

`IRef` is the class used to create interfaces and `localBind` is a function that creates a local binding between two interfaces. The interfaces and the local binding between object `a` and `b` in Figure 1 are created with the following Python code:

```
i = IRef(a,["f"],[])
j = IRef(b,[],["f"])
l = localBind(i,j)
r = j.f(2)
```

The arguments to the `IRef` constructor is (i) the object, (ii) the description of exported methods and (iii) the description of imported methods.[1] The `localBind` function returns a local binding control object that can be used to control the local binding. A local binding only exists as cross-references in the interfaces bound and is not dependent on its local binding control object (the local binding even exists after its control object is garbage collected). A local binding can be broken with the `breakBinding` function or with the `breakBinding` method of the local binding control object. The following code illustrates these two methods for the example above (select one):

```
breakBinding(i,j)
l.breakBinding()
```

An interface is accessed through an interface reference created with the `IRef` class. The interface reference contains a description of the interface including (i) its object, (ii) its exported methods, and (iii) its imported methods.

Specialised stream and signal interfaces are also available. These interfaces have a source and a sink pair. A local binding between a source and a sink is also created with the `localBind` function. The code below creates a stream interface reference source and sink pair and makes a local binding between them. The source interface is associated with object `a` and the sink interface is associated with object `b`:

```
src = StreamSrcIRef(a)
sink = StreamSinkIRef(b)
l = localBind(src,sink)
src.put(data)
```

[1]In this prototype the descriptions of exported and imported methods are lists of method names. This can be extended with an interface description language like CORBA IDL.

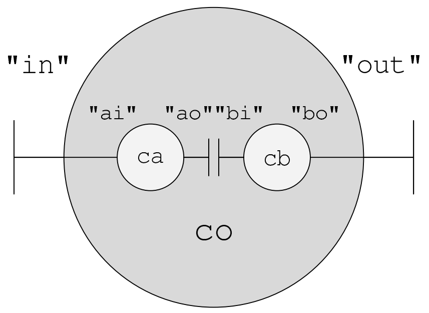


Figure 2. A composite component

A stream interface uses a `put` method with one data argument to insert data (a frame) into the stream. Object `b` in the example above implements the `put` method to receive data from the stream. A signal interface pair is created with the `SigSrcIRef` and the `SigSinkIRef` classes. The signal source interface provides (exports) an `event` method with no arguments for the actual transfer of a signal.

3.2 Components

Components are used to build both applications and the middleware in OOPP. A component is a unit of independent deployment [12]. They are developed and delivered independently and provide access to their requested and provided services (methods) through one or more specified interfaces. All interactions between components are specified through these well defined interfaces.

The interfaces of a component are available without any previous knowledge about the component providing them. The interfaces provided can be browsed and a specific interface can be accessed by its key (name). A primitive component that encapsulates one object and provides/requests access to/from its object through a set of interfaces can be created with the `Component` class. A component for object `a` in the example above can be created with the following statement:

```
ca = Component({"op":i},a)
```

The result is a component `ca` with an interface with key `"op"` represented by the interface reference `i` that exports the method `f`. The component encapsulates object `a` that implements method `f`. The interfaces of a component are available in the `interfaces` attribute[2] of the component. Interface `"op"` of `ca` (represented by the interface reference `i`) can be accessed with the `ca.interfaces["op"]` expression.

A composite component is a way to manage a complex component. In particular, a composite component encapsulates a graph of components. The outside view of a composite component is similar to a primitive component: it

[2]The `interface` attribute is a Python dictionary. A dictionary is similar to an associative array in Perl. The expression `{"op":i}` is a dictionary with one element with the key `"op"` and the value `i`.

provides a set of interfaces that can be browsed and accessed through their keys (names). Figure 2 illustrates a simple composite component `co` providing the external interfaces `"in"` and `"out"`. It contains two components `ca` and `cb` connected with a local binding between their interfaces `"ao"` and `"bi"`. The external interface `"in"` is a mapping to interface `"ai"` in component `ca` and the external interface `"out"` is a mapping to interface `"bo"` in component `cb`.

Composite components can be created with the generic `Composite` class. The composite component `co` in Figure 2 was created with the following statement:

```
co = Composite(
  {"in":(ca,"ai"),
   "out":(cb,"bo")},
  {"comps":[ca,cb],
   "iif":{"ao":(ca,"ao"),
          "bi":(cb,"bi")},
   "edges":[("ao","bi")]})
```

The first argument specifies the external interfaces of the component and the second the component graph.[3] The component graph is specified by the components contained in the composite component (`"comps"`), the set of internal interfaces (`"iif"`) and the set of edges (local bindings) between these internal interfaces (`"edges"`). The `Composite` class can be used to create any composite component. The result is a fairly complex constructor syntax. A composite component class is usually a refinement of the `Composite` class with a less complex constructor syntax. The binding objects discussed below are examples of such composite components.

3.3 Binding objects

Local bindings can only be used between interfaces in the same address space. In contrast, binding objects can be used to create bindings between interfaces in different address spaces or even on different nodes. A binding object replaces a local binding in such cases. An operational binding is a particular kind of binding object supporting operational interactions [23]. Binding objects for streams and signals are also available in OOPP.

Figure 3 illustrates how the built-in operational binding of OOPP is implemented. Capsule `A` and capsule `B` are two different address spaces. The operational binding is a composite component containing two components, `stub1` and `stub2`. It provides three external interfaces, one control interface `"ctrl"` and two connecting interfaces `"iface1"` and `"iface2"`. The two stubs are connected with two low-level TCP/IP bindings. One is for requests (method calls) from an object in capsule `A` to an object in capsule `B` (with reply back to the object in capsule `A`) and the other one is for requests from an object in capsule `B` to an object in capsule `A` (also with reply). Operational bindings are thus two-way bindings (method calls can go in both directions). The low-level TCP/IP binding are implemented with an `m` and an `l` object. The `m` objects are message objects (send request and receive reply) and the `l` objects are listen objects (receive request and send reply).

[3] In Python, dictionaries are written with curly braces, lists with square brackets, and tuples with normal parentheses. Elements are separated with commas.

An operational binding can be created with the constructor of the `OpBinding` class. However, since the binding is usually immediately used to connect two interfaces the `remoteBind` function can be used instead. The difference is that the `OpBinding` constructor only creates the binding object while the `remoteBind` function also binds the external interfaces of the binding object to the two interfaces to be bound (in the same way as `localBind` binds two bindings to be bound). Object `a` with interface `i` in capsule `A` exports its method `f`. Object `b` in capsule `B` with interface `j` imports method `f`. The following code creates an operational binding between interface `i` and `j` and then calls method `f` of object `a` in capsule `A` through interface `j` in capsule `B`:

```
b = remoteBind(i,j)
r = j.f(2)
```

The built-in signal binding provided by OOPP is also based on TCP/IP. A signal binding can be created with the constructor of the `SigBinding` class, but as argued above for `remoteBind` they can more easily be created with the `sigBind` function. The `sigBind` function creates a (remote) signal binding between a signal source interface and a signal sink interface.

The stream binding provided by OOPP is based on connected UDP/IP. It does not provide any retransmission, sequencing or buffering. Every data frame fed in to it from the source side will be delivered as quickly as possible to the sink side. This approach is too simple for many applications that need a more advanced stream binding. The stream binding provided, however, can be used as a starting point for the creation of a binding with the features needed for the given application. Stream bindings can be created with the constructor of the `StreamBinding` class or with the `streamBind` function. The `streamBind` function creates a stream binding between a stream source interface and a stream sink interface. The following code creates a stream binding between the stream source interface `src` and the stream sink interface `sink` and then sends 100 data frames through the binding to the sink:

```
b = streamBind(src,sink)
for i in range(100):
    src.put(data[i])
```

4 INFRASTRUCTURE

The infrastructure is a supporting environment for programs in OOPP. The infrastructure is influenced by the engineering viewpoint of RM-ODP [23].

Table 1 describes the different arguments and return values used in the descriptions of the services in the following text.

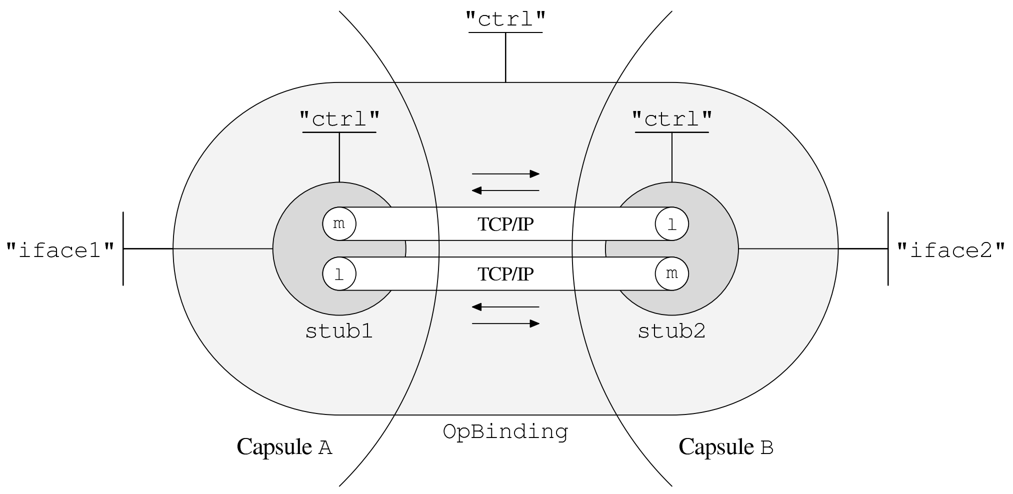


Figure 3. The implementation of an operational binding

i,j	Interface references
l	Local binding control object
c,d	Component instances
b	Binding object
u	Unique component identifier
C	Class or factory
r	Result value
k	Key or name
m	Method name
f	Method implementation
a	Argument tuple
w	Argument dictionary
I	Inspect dictionary
x*	x is optional

Table 1. Description of arguments and return values

4.1 Capsules

An OOPP address space is called a capsule. A capsule provides services for its local components (the components located in the address space). It can also provide services to remote components through a capsule proxy. A capsule proxy and a local capsule have identical interfaces, but requests through a capsule proxy are forwarded to the capsule it represents and replies are returned back to the caller through the proxy. The local capsule services is available through the `local` attribute of the `capsule` module. A capsule proxy for a remote capsule is created with the `CapsuleProxy` class. The following services represent the most common services provided by a capsule (and a capsule proxy):

```
registerComponent(c)→u
mkComponent(C,a*,w*)→u
delComponent(u)
getIRef(u,k)→i
localBind(i,j)→l
breakBinding(i,j)
callMethod(u,k,m,a*,w*)→r*
```

A component has to be registered in the capsule to be available for the services provided. A component created with the `mkComponent` method of the capsule will automatically be registered in the capsule. The `registerComponent` and `mkComponent` methods return a local unique identifier for the component (unique in the capsule). This identifier together with a capsule proxy provides a global identification of a component. The code below uses a capsule proxy `p` to create an instance of the component class `C` in a remote capsule (the constructor of class `C` is called without any arguments). The result is a component with the unique identifier `u`. It then creates an operational binding between the local interface `j` and the remote interface `i` of component `u`. It is then possible to call method `f` of the remote component `u` through the interface reference `j`.

```
u = p.mkComponent(C)
i = p.getIRef(u,"i")
b = remoteBind(i,j)
r = j.f(2)
```

The `getIRef` service of a capsule returns a global interface reference. In the example above the global interface reference of interface `"i"` in the remote component `u` is returned. The `getIRef` service of a capsule should always be used when a global interface reference of a registered component is needed.[4]

The capsule also provides services to establish and break a local binding in the capsule. These services are useful when a local binding in a remote capsule has to be established or broken. The request is then done through a capsule proxy.

[4]A global interface reference is useful outside the local capsule of the interface. It contains a 'unique component identifier' 'capsule proxy' pair to identify the component it is associated with (and its location).

The `callMethod` service of a capsule is a low-level method used to call a specific method in an interface of a registered component. This service is meant for implementing higher level bindings or services. The application programmer should use a binding object to establish a connection between two interfaces and call the remote method through the interfaces and the binding.

The following code illustrates the difference between using `callMethod` (line 1) and a operational binding (line 4). Method `f` is in both cases called with an argument `2` an the result is saved in `r`:

```
r = callMethod(u,"i","f",(2,))       1
i = p.getIRef(u,"i")                 2
b = remoteBind(i,j)                  3
r = j.f(2)                           4
```

The capsule also provides other low-level methods not listed above. This includes announcements (method calls without any return value), asynchronous (non-blocking) method calls and services to duplicate and move components. A capsule proxy can only be used to access a capsule if the capsule has started its serving loop. This has to be done explicitly with the `serve` method of the local capsule.

4.2 Name servers

A name service is needed to make it possible for components in different capsules to interact. The simple name service provided in OOPP is implemented with a name server. Interfaces and capsules can be registered (exported) and looked-up (imported) by a key (name). The name server is accessed through a name server proxy. A name server proxy for a given name server can be created with the knowledge of the location (the node) of the name server and the port the name server is listening on (a default port is used when it is not explicit given).

4.3 Factories

The task of a factory is to add new components to the environment [24]. An example of a complex factory is a stream binding factory. This factory has to create instances of stub components in different capsules. Other components like buffers and monitors might also have to be added. Some components might have to reserve resources and this could include negotiations. Because of the complexity of this task, factories are often specialised for the needs of a given application or system.

OOPP provides some generic factories for primitive and composite components. The syntax of the generic factory for composite components is complex. It has to contain a complete description of the new composite component. The `remoteBind`, `streamBind` and `sigBind` functions presented above are examples of specialised factories with a simpler syntax.

5 META-MODELS

OOPP implements two of the Open-ORB meta-models: the encapsulation and the composition meta-model. An implementation of the resource meta-model is presented in [17] and [25].

5.1 Encapsulation

The encapsulation meta-model provides access to the representation or the implementation of interfaces and objects (components). The encapsulation meta-model of a given object or interface is accessed through its encapsulation meta-object. A meta-object does not exist until it is accessed. The `encapsulation` function is used to get access to the encapsulation meta-object of an object or an interface. This is the most common services of the encapsulation meta-object:

```
inspect()→I
addMethod(k,f)
addXxxMethod(k,f)
addGetAttr(k,f)†
addSetAttr(k,f)†
changeClass(C)†
addImpMethod(k)‡
addImpXxxMethod(k,f)‡
changeObject(n)‡
restore()
```

The services marked with † are only available from the meta-objects of objects and services marked with ‡ are only available from the meta-object of interfaces. The services listed without any marks have effect on the exported methods when they are applied on interfaces.

The `inspect` method returns a detailed description of the object or the interface. The `restore` method removes the encapsulation meta-object.

New methods can be added to an object or to the exported methods of an interface with the `addMethod` service. A new method `get` that returns the value of the attribute `x` is added to object `o` with the following code:

```
def getx(self):
    return self.x
eo = encapsulation(o)
eo.addMethod("get",getx)
print o.get()
```

The add*Xxx*Method listed above represents the three methods `addPreMethod`, `addPostMethod` and `addWrapMethod` that are used to add pre-, post- and wrap-methods, respectively. A pre-method is a method that is called before the actual method is called. The pre-method has access to the arguments of the method and it can read and change their values. A post-method is a method that is performed after the actual method has returned. It has access to the arguments and the return value of the method and it can change the return value before it is passed to the caller. A wrap-method is wrapping the actual method. It

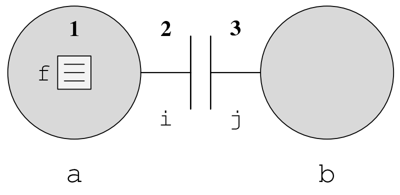


Figure 4. Different places to manipulate methods

can manipulate the arguments and the return values. It can even decide not to call the actual method at all. Below is a post-method added to the `get` method of object `o`. The post-method prints the return value of `get` before it returns to the caller:

```
def p(self,m):
    print m["result"]
eo = encapsulation(o)
eo.addPostMethod("get",p)
```

The function `p` in the code above has two arguments. All pre-, post- and wrap-methods have to have these two arguments. The first argument `self` is a reference to the object (as in every method of an object in Python). The second argument `m` is a dictionary containing information about the method call including the arguments and the return value (`"result"`).

The `addImpMethod` is used to add a method to the list of imported methods of an interface. The usage of `addImpMethod` can be demonstrated together with the `addMethod` service for interfaces (the semantics of the `addMethod` service for interfaces are "add an exported method to the interface"). Object `o` has an interface `o.i` that exports some methods and is bound with a local binding to another interface `j`. The code below adds method `get` from object `o` to these interfaces. Notice that `addImpMethod` also updates the local binding.

```
ei = encapsulation(o.i)
ei.addMethod("get",o.get)
ej = encapsulation(j)
ej.addImpMethod("get")
print j.get()
```

Figure 4 illustrates the different places where pre-, post-, and wrap-methods can be added. The example contains an object `a` with a method `f`. Interface `i` exports method `f` and interface `j` imports method `f`. Interface `i` and `j` are connected with a local binding. The encapsulation meta-object `ea` for `a` can be used to add pre-, post- and wrap-methods to `f` in object `a` (**1** in Figure 4). The encapsulation meta-object `ei` for `i` can be used to add pre-, post- and wrap-methods to the exported method `f` in interface `i` (**2** in Figure 4). Finally, the encapsulation meta-object `ej` for `j` can be used to add pre-methods, post-methods and wrap-methods to the imported method `f` in interface `j` (**3** in Figure 4). The example below adds a post-method at each place **1** (line 4), **2** (line 5) and **3** (line 6) in Figure 4. All the added post-methods print out the return value (`"result"`) before they increase it with one.

```
def inc(self,m):                          1
    print m['result'],                    2
    m['result'] = m['result'] + 1         3
ea.addPostMethod('f',inc)                 4
ei.addPostMethod('f',inc)                 5
ej.addImpPostMethod('f',inc)              6
print j.f(2)                              7
```

Suppose method `f` with argument 2 in object `a` returns the value 1. The output from the code above will then be "`1 2 3 4`", where 4 is the final return value.

Objects have attributes and the encapsulation meta-object can install methods to be called when the attributes of an object are read or changed. These features are useful for range checks and different monitoring tasks. One possibility is to monitor when a given attribute becomes greater than a limit value and then raise an alarm.

5.2 Composition

Complex binding objects are typical composite components. Multimedia applications in mobile environments are examples where the component graph of composite components need to be manipulated and restructured (changed and extended) during their life cycle. The composition meta-model is provided for this kind of manipulation of composite components.

The composition meta-model of a composite component is accessed through its composition meta-object. The `composition` function is used to get access to the composition meta-object of a composite component. The services provided are influenced by the operations originally proposed in the Adapt project for the manipulation of object graphs of open bindings [26]. The following operations are available from the composition meta-object:

```
inspect()→I
add(c)
remove(c)
bind(i,j)
break(i,j)
replace(c,d)
```

The `inspect` method of a composition meta-object returns a description of the composite component including the contained objects and the edges of the component graph. The `add` method adds a new component to the composite component, but no new edges (local bindings) are created. The new component can be located in a remote capsule. The `bind` method creates a new edge in the component graph and the `break` method breaks such a binding. It is transparent for the user if the actual local binding is created or removed locally or in a remote capsule. The `replace` method replaces an existing component with a new one. The new component must have a matching set of interfaces.

6 QOS MANAGEMENT

OOPP includes support for quality of service (QoS) management. The management is done with a set of management objects connected with signal bindings. A management object can use the meta-space of the components under management to change the behaviour of these components. This is done to fulfil the goal of the given management policy.

A management object can have three different roles. A *monitor* collects and filters information from the running system. For example, a monitor of a buffer could check if buffer overflow occurs to often (with a given definition of 'to often'). A *strategy selector* collects information from the monitors and, based on this information and on timing constraints, decide to select a management strategy. For example, a strategy selector that receives a buffer "overflow to often" signal from a monitor could decide to delay the stream at its source. The *strategy activator* activates (performs) the selected strategy when it receives a signal from a strategy selector. The monitors and the strategy activators can use the meta-spaces of the components under management to perform their tasks.

Figure 5 illustrates a management setup for a producer/consumer case. The monitor `M` monitors if there is a buffer "overflow to often". The strategy selector `S` selects strategy `P` or `C`. Strategy activator `C` manipulates the meta-models of the consumer and strategy activator `P` manipulates the meta-models of the producer.

The monitors and strategy selectors are implemented with automata components. Their behaviour is specified in formal timed automata descriptions. One of the advantages of using automata is that we can reason about and simulate their behaviour. This, together with a formal description of the complete system can be used to simulate and formally reason about the whole system [27, 28].

An example of the usage of this QoS management approach is presented in [29]. In this example an audio stream dynamically adapts to the variable QoS delivered by the network. This could be further extended with resource management using the resource meta-model presented in [17].

A more detailed quality of service management example can be found in . This also includes a detailed evaluation of the approach.

7 RELATED WORK

There is growing interest in the use of reflection in distributed systems. Pioneering work in this general area was carried out by McAffer [11]. With respect to middleware in particular, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [30, 31], focusing on reification of invocation marshalling and dispatching. In addition, researchers at APM have developed an experimental middleware platform called FlexiNet [32], which allows the programmer to tailor the underlying communications infrastructure by inserting/removing layers. Their solution is, however, language-specific, i.e. applications must be written in Java. Researchers at DSTC have developed a distributed infrastructure for mobile computing [33]. This infrastructure includes adaption rules to sense and react to environment variations. Manola has carried out work in the design of a "RISC" object model for distributed computing [34], i.e. a minimal object model which can be specialised through reflection. Finally, researchers at the Ecole des Mines de Nantes are also investigating the use of reflection in proxy mechanisms for ORBs [35].

Our design has been influenced by a number of specific reflective languages. The concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [15]. Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [8] and CodA [11]. Both these systems feature decompositions of meta-space in terms of the acceptance of messages, placing the message in a queue, their selection, and the subsequent dispatching.

8 FUTURE WORK

OOPP currently only implements the encapsulation and the composition meta-models. One obvious task for future work is to implement the environment and the resource meta-models.

An implementation of the environment meta-model could learn a lot from ATOM [36]. The separation of the concurrent execution and synchronisation features from the object-oriented features in the ATOM programming model has a lot in common with the distinct environment meta-model in Open-ORB. The implementation of the resource meta-model should be based on the work presented in [17] and [25].

A tool to manipulate and deploy the components of an application in a graphical user interface (GUI) would increase the power of this programming environment. Such a tool should also have the possibility to access the meta models of the interfaces and components.

Other future tasks include the implementation of a generic event notification service for components and a richer set of (stream) binding objects, and adding type information and type checks for the interfaces. Security and performance are other obvious tasks ignored so far.

9 CONCLUDING REMARKS

This paper has presented OOPP, the Open-ORB Python Prototype. OOPP implements a reflective component-based middleware platform with support for quality of service management. The paper focused on the programming model and the meta-models provided by OOPP. OOPP implements an RM-ODP like programming model where the configurability and openness is provided in a principled way via the concept of reflection. Applications including multimedia, real-time requirements and mobile computing

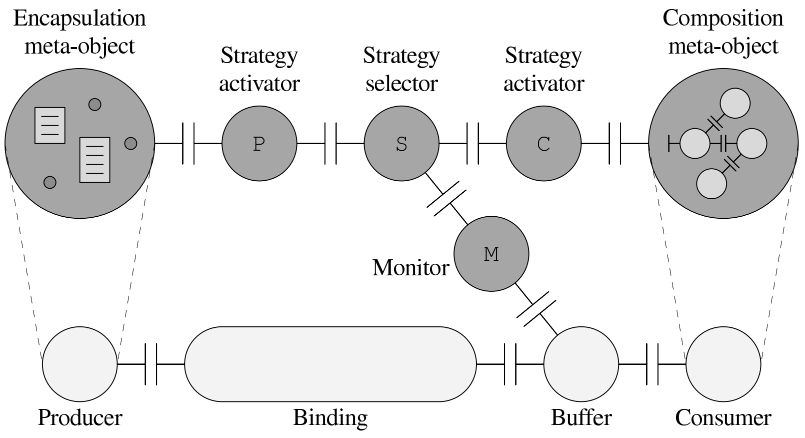


Figure 5. A management setup for the producer/consumer case

seem to benefit from this approach. Deployable components are the foundation for the flexibility (reconfiguration) provided by the system.

Experiences show that OOPP can successfully be used in the development of a wide variety of applications. Performance has not been a primary concern in current OOPP. However, the prototype has been able to deliver the required performance in audio stream examples with reasonable complex quality of service management [29].

The current version of OOPP is available from the Python starship.[5]

ACKNOWLEDGEMENTS

Michael Papathomas has contributed a lot to the early attempts to introduce reflection to the programming model. Fabio Costa has also been an important discussion partner in the realisation of the current programming model and meta-models. David Sánchez had the pleasure of being the important first user of OOPP. His feedback has been valuable for the actual implementation. Lynne Blair has contributed a lot to the automata-based quality of service management implemented in OOPP. Geoff Coulson has asked the difficult questions and contributed a lot to the development and understanding of the meta-models. Finally, thank you to the various members of the Distributed Multimedia Research Group at Lancaster University, the Information and Communication Technology group at NORUT IT, and the Department of Computer Science at University of Tromsø that contributed to discussions on areas related to this work.

REFERENCES

[1] P. A. Bernstein, "Middleware: A model for distributed system services," *Communications of the ACM*, vol. 39, pp. 86–98, Feb. 1996.

[2] W. W. Eckerson, "Three-tier client/server architecture," *Open Information Systems*, vol. 10, pp. 3–22, Jan. 1995.

[3] S. S. King, "Middleware!," *Data Communications*, vol. 21, pp. 58–67, Mar. 1992.

[4] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Middleware'98*, Sept. 1998.

[5] F. Costa, G. Blair, and G. Coulson, "Experiments with reflective middleware," in *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, vol. 1543 of *Lecture Notes in Computer Science*, (Brussels, Belgium), pp. 390–391, July 1998.

[6] B. C. Smith, *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[7] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[8] T. Watanabe and A. Yonezawa, "Reflection in an object-oriented concurrent language," in *OOPSLA '88 Proceeding*, vol. 28 of *Sigplan Notices*, pp. 306–315, ACM Press, Sept. 1988.

[9] G. Agha, "The structure and semantics of actor languages," in *The Proceedings of REX School/Workshop Foundations of Object-Oriented Languages* (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), vol. 489 of *Lecture Notes in Computer Science*, (Noordwijkerhout, The Netherlands), pp. 1–59, Springer-Verlag, May/June 1990.

[10] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation," in *Proceedingss of OOPSLA'92*, vol. 28 of *Sigplan Notices*, pp. 414–434, ACM Press, 1992.

[5]http://starship.python.net/crew/anders/oopp/

[11] J. McAffer, "Meta-level architecture support for distributed objects," in *Proceedings of Reflection'96* (G. Kiczales, ed.), (San Francisco), pp. 39–62, 1996.

[12] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, 1998.

[13] ISO/IEC, "Open distributed processing reference model, part 2: Foundations," ITU-T Rec. X.902 — ISO/IEC 10746-2, ISO/IEC, 1995.

[14] K. B. Saikoski and G. Coulson, "Adaptive groups in Open ORB," in *Proceedigns of the CAiSE '99 Doctoral Consortium*, (Heidelberg, Germany), June 1999.

[15] H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A distributed programming system with multi-model reflection framework," in *Proceedings of the Workshop on New Models for Software Architecture*, Nov. 1992.

[16] G. S. Blair, F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and J.-B. Stefani, "The design of a resource-aware reflective middleware architecture," in *Proceeding of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, vol. 1616 of *Lecture Notes in Computer Science*, (St-Malo, France), pp. 115–134, Springer-Verlag, 1999.

[17] H. A. Duran and G. Blair, "A resource management framework for adaptive middleware," in *3th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, (Newport Beach, California, USA), Mar. 2000.

[18] G. van Rossum, "Glue it all together with Python," in *Workshop on Compositional Software Architectures*, (Monterey, California), Jan. 1998. (position paper).

[19] M. Lutz, *Programming Python*. A Nutshell Handbook, O'Reilly & Associates, Inc., 1996.

[20] K. Farooqui, L. Logrippo, and J. de Meer, "The ISO reference model for open distributed processing: an introduction," *Computer Networks and ISDN Systems*, vol. 27, pp. 1215–1229, July 1995.

[21] ISO/IEC, "Open distributed processing reference model, part 1: Overview," ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC, 1995.

[22] A. Andersen, "A note on reflection in Python 1.5," Distributed Multimedia Research Group Report MPG-98-05, Lancaster University, UK, Mar. 1998.

[23] ISO/IEC, "Open distributed processing reference model, part 3: Architecture," ITU-T Rec. X.903 — ISO/IEC 10746-3, ISO/IEC, 1995.

[24] G. Blair and J.-B. Stefani, *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[25] H. A. Duran and G. Blair, "Configuring and reconfiguring resources in middleware," in *1st International Symposium on Advanced Distributed Systems*, (Guadalajara, Jalisco, Mexico), Mar. 2000.

[26] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting adaptive multimedia applications through open bindings," in *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98)*, (Annapolis, Maryland, US), May 1998.

[27] L. Blair, *The Formal Specification and Verification of Distributed Multimedia Systems*. Dr. thesis, Department of Computer Science, Lancaster University, Sept. 1994.

[28] G. Blair, L. Blair, H. Bowman, and A. Chetwynd, *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.

[29] D. S. Gancedo, "QoSMonAuTA, QoS monitoring and adaptation using timed automata," Master's thesis, Lancaster University, UK, Sept. 1999.

[30] A. Singhai, A. Sane, and R. Campbell, "Reflective ORBs: Supporting robust, time-critical distribution," in *ECOOP'97 Workshops — Reflective Real-Time Object-Oriented Programming and Systems* (J. Bosch and S. Mitchell, eds.), vol. 1357 of *Lecture Notes in Computer Science*, (Jyväskylä, Finland), pp. 55–61, Springer-Verlag, June 1997.

[31] M. Roman, F. Kon, and R. H. Campbell, "Design and implementation of runtime reflection in communication middleware: the dynamicTAO case," in *ICDCS'99 Workshop on Middleware*, (Austin, Texas), May 1999.

[32] R. Hayton, "FlexiNet open ORB framework," Tech. Rep. APM.2047.01.00, ANSA Phase III, APM, 1997.

[33] A. Rakotonirainy and M. Chilvers, "A distributed infrastructure for mobile computing," in *The Sixth International Python Conference*, (San Jose, California), pp. 89–95, Oct. 1997.

[34] F. Manola, "Meta object protocol concepts for a "RISC" object model," Tech. Rep. TR-0244-12-93-165, GTE Laboratories, 1993.

[35] T. Ledoux, "Implementing proxy objects in a reflective ORB," in *Procceings of the ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, (Jyväskylä), 1997.

[36] M. Papathomas and A. Andersen, "Concurrent objectorieted programming in Python with ATOM," in *The Sixth International Python Conference*, (San Jose, California), pp. 77–87, Oct. 1997.

C.2 IEE Proceedings – Software 147(1)

Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform

Gordon S. Blair, Anders Andersen*, Lynne Blair, Geoff Coulson and David Sánchez**

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg,
Lancaster, LA1 4YR,
U.K.

E-mail: [gordon, lb, geoff]@comp.lancs.ac.uk

* aa@computer.org

** david.sanchez@switzerland.org

Abstract

Reflection has recently been applied in a variety of settings to introduce more openness and flexibility into designs. This paper builds on our previous work in applying reflection to the design of middleware platforms by considering the role of reflection in supporting the dynamic QoS management functions of monitoring and adaptation. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies. A pilot implementation of our QoS management scheme is discussed and evaluated and examples of the use of our approach are given.

1. INTRODUCTION

Reflection initially emerged in the programming language community as an important technique to introduce more flexibility and openness into the design of programming languages. In recent years, researchers have studied the potential impact of reflection in other areas including windowing systems, operating systems and file systems. Again, the motivation has been to introduce more flexibility and openness in the designs. The authors believe that the same technique can usefully be exploited in the area of distributed systems, in general, and middleware[1], in particular. For example, reflection can help provide the necessary level of adaptability required by emerging distributed systems application areas such as multimedia and mobile computing.

In previous papers, the authors have reported on the design and implementation of reflective middleware platforms [Blair98, Costa98]. The aim of this paper is to consider the implications of such an architecture for the area of Quality of Service (QoS) management. More specifically, the paper describes how critical QoS management functions can be incorporated into our reflective middleware platform, with particular emphasis on the *dynamic* QoS management functions of QoS monitoring and adaptation. It is argued that the reflective approach leads to a natural and highly flexible implementation of such QoS management functions (and indeed the approach offers considerable benefits over more conventional approaches).

The paper is structured as follows. Section 2 presents the case for reflective middleware and section 3 summarises the main features of our architecture. Section 4 then examines our approach to

[1] We use the term *middleware* to refer to distributed systems platforms that offer a language and platform independent programming model, located between the application and the underlying distributed environment. Examples of middleware platforms include the OMG's CORBA [OMG98a] and Microsoft's DCOM.

QoS management, highlighting the role of timed automata in realising key QoS management functions. Consideration is also given to the issue of formal verification of QoS management subsystems. Following this, section 5 discusses and evaluates our pilot implementation and presents examples of our approach, illustrating the role of reflection in supporting QoS management policies. Finally, section 6 discusses related work and section 7 presents some concluding remarks.

2. THE CASE FOR REFLECTIVE MIDDLEWARE

The concept of reflection was first introduced by Smith in 1982 [Smith82]. In this work, he introduced the reflection hypothesis which states:

> *"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures".*

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a *meta-object protocol (MOP)* which defines the services available at the *meta-level*. Examples of operations available at the meta-level include altering the semantics of message passing and inserting before or after actions around method invocations. Access to the meta-level is provided through a process of *reification*. Reification effectively makes some aspect of the internal representation explicit and hence accessible from the program. The opposite process is then *absorption* where some aspect of meta-system is altered or overridden.

Smith's insight has catalysed a large body of research in the application of reflection. Initially, this work was restricted to the field of programming language design [Kiczales91, Watanabe88, Agha91]. More recently, the work has diversified with reflection being applied in operating systems [Yokote92] and, more recently, distributed systems (see section 6).

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open engineering. For example, reflection can be used to *inspect* the internal behaviour of a language or system. By exposing the underlying implementation, it becomes straightforward to insert additional behaviour to monitor the implementation, e.g. performance monitors, quality of service monitors, or accounting systems. Reflection can also be used to *adapt* the internal behaviour of a language or system. Examples include replacing the implementation of message passing to operate more optimally over a wireless link, introducing an additional level of distribution transparency in a running computation (such as migration transparency), or inserting a filter to reduce the bandwidth requirements of a communications stream. Although reflection is a promising technique, there are a number of potential drawbacks of this approach; in particular issues of *integrity* and *performance* must be carefully addressed (we return to these issues in sections 3.3 and 7 respectively).

In contrast, the present-day approach to developing middleware platforms is generally to adopt a *black box* philosophy, whereby implementation details are hidden from the platform user (cf. distribution transparency). There is increasing evidence though that the black box philosophy is becoming untenable. For example, the OMG have recently added internal interfaces to CORBA to support services such as transactions and security. The recently defined Portable Object Adapter is another attempt to introduce more openness in their design. Nevertheless, their overall approach can be criticised for being rather ad hoc. Similarly, a number of ORB vendors have felt obliged to expose selected aspects of the underlying system (e.g. filters in Orbix or interceptors in COOL). These are however non-standard and hence compromise the portability of CORBA applications and services. The RM-ODP architecture partially addresses this issue by distinguishing between computational and engineering concerns. However, the authors believe this approach is still not sufficient to meet the demands of the next generation of distributed applications (see section 3.3 below).

The authors believe that the solution is to provide flexible middleware platforms through application of the principle of reflection.

3. AN ARCHITECTURE FOR REFLECTIVE MIDDLEWARE

3.1. General Principles

In our reflective architecture, we adopt a *component-based* model of computation [Szyperski98]. A middleware platform is then viewed as a particular configuration of components, which can be selected at build-time and re-configured at run-time. We therefore provide an open and extensible library of components, and component factories, supporting the construction of such platforms, e.g. protocol components, schedulers, etc. The use of components is important given the trend towards the application of this technology in open distributed processing, e.g. CORBA v3 [OMG99] and Microsoft's DCOM. Note however that these technologies exploit component technology at the application level; we extend this approach to the structuring of the middleware platform itself. Our particular component model includes features to support multimedia applications, and is derived from previous work on the Computational Model from RM-ODP. The main features of our component model are: i) components are described in terms of a set of *required* and *provided* interfaces, ii) interfaces for *continuous media* interaction are supported, iii) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding component*), and iv) components offer a built-in *event notification service*. The component model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details on the underlying RM-ODP Computational Model can be found in [Blair97]. In contrast, with RM-ODP, however, we adopt a consistent computational model throughout the design (see section 3.3).

As our second principle, we adopt a *procedural* (as opposed to a declarative) approach to reflection, i.e. the meta-level (selectively) exposes the actual "program" (in our case, configuration of components) that implements the system. In our context, this approach has a number of advantages over the declarative approach. For example, the procedural approach is more primitive (in particular, it is possible to support declarative interfaces on top of procedural reflection but not vice versa). Procedural reflection also opens the possibility of an infinite tower of reflection (i.e. the base level has a meta-level, the meta-level is implemented using components and hence has a meta-meta-level, and so on). This is realised in our approach by allowing such an infinite structure to exist *in principle* but only to instantiate a given level *on demand*, i.e. when it is reified. This provides a finite representation of an infinite structure (a similar approach is taken in ABCL/R [Watanabe88]). Access to different meta-levels is important in our design although in practice most access will be restricted to the meta- and meta-meta-levels (e.g. see the examples in section 5.2).

The third principle underlying our design is to support *per interface* (or, sometimes, *per component*) meta-spaces. This is necessary in a heterogeneous environment where components will have varying capacities for reflection. Such a solution also provides a fine level of control over the support provided by the middleware platform; a corollary of this is that problems of maintaining integrity are minimised due to the limited scope of change (see discussion in section 3.3). We do recognise however that there are situations where it is useful to be able to access the meta-spaces of sets of components in one single action; to support this, we also allow the use of *groups* [Saikoski99] (we do not address this aspect further in this paper however).

The final principle is to structure meta-space as a number of closely related but distinct *meta-space models*. This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [Okamura92]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. Further details of each of the various models can be found below.

3.2. The Design of Meta-space

3.2.1. Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content of a given component [Watanabe88]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the *encapsulation* and *composition* meta-models. We introduce each model in turn below.

The *encapsulation meta-model* provides access to the representation of a particular interface in terms of its set of methods and associated attributes, together with key properties of the interface including its inheritance structure (where available). This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. Clearly, however, the level of access provided by the encapsulation model will be language dependent. For example, with compiled languages such as C, access may be limited to introspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python (see section 5.1), more complete access is possible, such as being able to add or delete methods and attributes. This level of heterogeneity is supported by having a type hierarchy of encapsulation meta-interfaces ranging from minimal access to full reflective access to interfaces. Note however that it is important that this type hierarchy is open and extensible to accommodate unanticipated levels of access.

In reality, many components will in fact be *composite*, using a number of other components in their construction. In recognition of this fact, we also provide a *compositional meta-model* offering access to such constituent components. Note that this meta-model is associated with each component and not each interface. More specifically, the same compositional meta-model will be reached from each interface defined on the component (reflecting the fact that it is the component itself, rather than the interface, that is composite). In the meta-model, the composition of a component is represented as a *component graph*, in which the constituent components are connected together by *local bindings*[2]. The interface offered by the meta-model then supports operations to inspect and adapt the graph structure, i.e. to view the structure of the graph, to access individual components in the graph, and to adapt the graph structure and content.

This meta-model is particularly useful when dealing with binding components [Blair98]. In this context, the composition meta-model reifies the internal structure of the binding in terms of the components used to realise the end-to-end communication path. For example the component graph could feature an MPEG compressor and decompressor and an RTP binding component. The structure can also be exposed recursively; for example, the composition meta-model of the RTP binding might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol. It is argued in [Fitzpatrick98] that open bindings alone provide strong support for mobile computing.

3.2.2. Supporting Behavioural Reflection

Behavioural reflection is concerned with activity in the underlying system [Watanabe88]. This is represented by a single meta-model associated with each interface, the *environmental meta-model*. In terms of middleware, the underlying activity equates to functions such as message arrival, enqueueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [Watanabe88, McAffer96].

Again, different levels of access are supported. For example, a simple meta-model may enable the insertion of pre- and post- methods. Such a mechanism can be used to introduce, for example, additional levels of distribution transparency (such as concurrency control) or to insert functions such as security managers or compression components. A more complex meta-model may allow the inspection or adaptation of each element of the processing of messages as described above. With such complexity, it is likely that such a meta-model would itself be a composite component with individual components accessed via the compositional meta-space of the environmental meta-space. As with the other meta-models, heterogeneity is accommodated within an open and extensible type hierarchy.

3.2.3. Supporting Resource Access

Most reflective languages and systems restrict their scope to the above styles of reflection. In experiments, however, we have identified a significant weakness of this approach, namely that we have no means of accessing the level of resources and resource management in the system. This is a particular problem for mobile, multimedia and real-time systems where it is often important to be aware of the resources currently available at a given node (e.g. if it is intended to introduce a new software compression component). We therefore introduce a fourth meta-model, referred to as the *resource meta-model* [Blair99b]. This meta-model supports the reification of resource creation, scheduling and, more generally, management. It is important to stress that this meta-model is fully orthogonal to the others. For

2 The RM-ODP inspired concept of local binding is crucial in our design, providing a language-independent means of implementing the interaction point between interfaces.

example, the environmental meta-model identifies the steps involved in processing an arriving message; the resources meta-model then identifies the resources required to perform this processing. More generally, the resources meta-model is concerned with the allocation and management of resources associated with any activity in the system.

The meta-model is based around the abstractions of *resources* and *tasks*[3]. Resources can be either primitive (e.g. raw memory or OS threads) or complex (e.g. buffers or user-level threads multiplexed on OS threads). They are created and managed by *resource factories* and *resource managers* which typically build complex resources by adding value to, or combining, primitive resource instances. For example, a user level scheduler is a resource manager which builds user level threads from OS threads. Both resources and their managers are uniformly viewed as components. Tasks are then the logical unit of activity in the system with the precise granularity varying from configuration to configuration. For example, there could be a single task dealing with the arrival, filtering and presentation of an incoming video stream, or alternatively this could be divided into a number of smaller tasks. Importantly, tasks can span component boundaries and are thus orthogonal to the structure of the system. Tasks are essentially the unit of resource allocation, i.e. tasks have a pool of resources to carry out their desired activity. Tasks also have associated task managers (analogous to the services identified above for resources).

There is a resources meta-model *per address space*, i.e. resources are associated with a particular address space and all components within that address space share the same meta-model. The meta-model provides access to a set of components representing resources, together with the associated managers. As with other meta-models, it is then possible to either inspect or adapt activity associated with resources. For example, it is possible to insert monitors to capture statistics on the effectiveness of a thread scheduling policy and then possibly change this policy based on the information collected. In programming terms, the resources meta-model is accessed in a similar way to the compositional meta-model; i.e. as a graph structure which organises resources, tasks and managers into hierarchical structures.

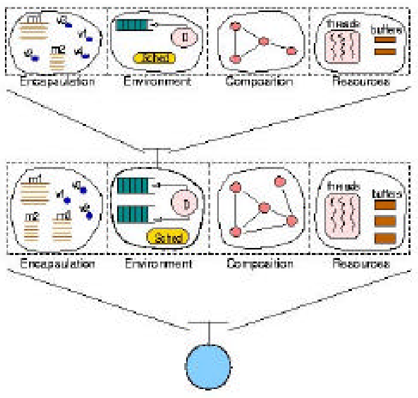


Figure 1: The structure of meta-space.

The complete architecture is summarised in figure 1.

3.3. Discussion

In our opinion, the reflective architecture described above provides a promising basis for the design of future middleware platforms and overcomes certain inherent limitations of technologies such as CORBA. In particular, the architecture offers principled and comprehensive access to the engineering of a middleware platform. This compares favourably with CORBA which, as stated above, generally follows a "black box" philosophy with minimal, ad hoc access to internal details.

We also believe that the reflective approach generalises the *viewpoints* approach to structuring advocated by RM-ODP. RM-ODP distinguishes between the Computational Viewpoint (focusing on application-level objects and their interaction) and the Engineering Viewpoint (which considers their implementation in a distributed environment). Crucially, each viewpoint also has its own set of object modelling concepts (for example, the Computational Viewpoint features objects, interfaces and bindings, whereas the Engineering Viewpoint has basic engineering objects, capsules and protocol objects). Consequently, this approach enforces a two-level structure, i.e. it is not possible to analyse engineering objects in terms of their internal structure

[3] This aspect of the work is being developed along with our collaborators at CNET, France Telecom. In particular, they have developed the particular resource/ task model presented in this section.

or behaviour. Our approach overcomes this limitation by offering a consistent object (component) model throughout, supporting arbitrary levels of openness.

Another benefit of our approach is that it reduces problems of maintaining integrity. This is due to our approach to scoping whereby every component/interface has its own meta-space. Thus changes to a meta-space can only affect a single component. Furthermore, the meta-space is highly structured, again minimising the scope of changes. An additional level of safety is provided by the strongly typed component model.

4. INTRODUCING QOS MANAGEMENT

4.1. General Approach

As can be seen from the descriptions above, the concept of component graphs is central to our design. In particular, the composition, environment and resource meta-models are all represented as component graphs. Given this, QoS management is largely concerned with:

i) creating the initial configurations of components and resources based on a statement of desired QoS properties and the assumed QoS from the environment (e.g. selecting appropriate compression components to be used), and

ii) inspecting and adapting the corresponding component graphs depending on the actual QoS attained and the actual QoS offered by the environment (which might of course change in, for example, a mobile environment).

The first aspect is referred to as *static* QoS management and the second aspect as *dynamic* QoS management. As stated earlier, we focus our attention on the dynamic aspects of QoS management in this paper (the static aspects are the responsibility of the various factories, including binding factories, and are the subject of ongoing research). Dynamic QoS management is achieved by introducing *management components* into the component graph structure (accessed via meta-space). To maintain a clean separation of concerns between management components and components being managed, communication between the two is achieved by an *event notification* mechanism (as provided by our component model; see section 3.1). Separation is achieved because components do not need to know in advance if they are to be managed. In other words, managers can be introduced at any time and can then register for events of interest (and receive call-backs when the specific events occur).

Different styles of management component are identified in our architecture (see table 1).

Monitoring	
Event Collector	Observe behaviour of underlying functional components and generate relevant QoS events.
Monitor	Collect QoS events and report abnormal behaviour to interested parties.
Control	
Strategy Selectors	Select an appropriate adaptation strategy (i.e. strategy activator) based on feedback from monitors.
Strategy Activators	Implement a particular strategy, e.g. by manipulating component graph.

Table 1: Styles of management component.

The role of *monitoring* is to collect statistics on the level of QoS attained by the running system and to raise events when problems occur, e.g. to collect information on the latency and throughput of a video presentation and raise exceptions should they fall outside given thresholds. This is sub-divided into event collectors, which interface with the underlying implementation, and monitors which detect QoS violations. *Control* is then responsible for implementing adaptation policies in response to such events. We distinguish between *strategy selectors* and *strategy activators* (which together realise the adaptation policy). Strategy selectors decide on which approach should be taken in response to QoS degradation, e.g. degrading the quality of the video presentation or providing additional resources. In contrast, strategy activators are responsible for the detailed implementation of this strategy, typically by manipulating the component graph, e.g. introducing a jitter compensation buffer or accessing and adapting resources through the resource meta-model. The rationale for this division is to provide a cleaner architecture and to promote re-use of management components. In addition, it is useful to distinguish between the implementation-oriented aspects of QoS management (i.e. event collectors and strategy activators), and the policy-oriented aspects (i.e. monitors and strategy selectors), and perhaps use a different languages for each aspect (see section 4.2.1 below). It should be stressed though that, in implementation, the different functions can be composed together (we expand this aspect of composition in section 4.2.2 below).

At a first glance, this might appear to be a fairly traditional approach to QoS management. However, when combined with the capabilities of a reflective architecture, some interesting properties emerge. Firstly, the approach is completely dynamic. New management components can be introduced at any time and at any place in the underlying configuration. Similarly, they can be removed when no longer needed. Both these actions are initiated by re-configuring component graphs. Secondly, the policy for management is itself open to inspection and adaptation through reification of management components. To enable this, we assume that management components consist of a policy written in an appropriate scripting language, together with an in-built interpreter for that scripting language (see section 5.2).

The architecture is open in that any scripting language can be used, although we do provide support for one particular language (see section 4.2). Reflection can then be used to access or modify this policy, e.g. by downloading a new management script or altering some parameters for the script. More specifically, we can introduce meta-managers (again consisting of monitors and controllers) to effectively manage the management structure. Whereas managers implement policy, meta-managers implement meta-policy. As an example, consider the use of header compression in a low bandwidth environment. A manager could have the role of switching header compression on or off based on a given bandwidth threshold (an attribute of the manager component). A meta-manager would then be able to alter this threshold depending on the current processor load. We argue that this is a useful facility in highly dynamic environments.

4.2. QoS Management Policies

4.2.1. Specification of Policies

As stated above, the architecture is open in that management components can be written using any scripting language. We do however provide support for one particular notation for the specification of QoS management policies, namely *timed automata*. We typically use timed automata for the specification of monitors and strategy selectors, but not necessarily for event collectors and strategy activators (especially, for example, for complex strategies requiring significant manipulation of the component graph structure).

We define our timed automata formally as:

$$TA = \langle S, s_0, \rightarrow, I \rangle$$

where S is a finite set of states,
$s_0 \in S$ is the initial state,
$\rightarrow$ is a transition relation, and
I is an invariant assignment function on a given state; all variables must satisfy the invariant whilst the automaton is operating in this state.

Transition relations have the form:

$$\rightarrow = \langle l, g, a, r, l' \rangle$$

where l, l' are nodes of the automaton,
a are actions,
g is a guard, and
r is a reset (or reassignment) function.

The transitions must satisfy certain rules detailed in [Blair99a]; these only allow transitions to occur when guards are true and ensure that the required variables are reset on transitions. Furthermore, time can only pass in a given state if the invariant at the new time holds true.

Examples of using timed automata can be found in section 5.2 below.

4.2.2. Formal Verification

One benefit of adopting timed automata is that it is then possible to *verify* QoS management functions. Indeed, we can go further and verify QoS management subsystems plus their interactions with the rest of the system. To support this, we are developing a *multi-paradigm specification* technique whereby different aspects of the system can be written in different (formal) notations. For example, the expected behaviour of the system can be specified using languages such as LOTOS, with QoS management functions specified using timed automata. We also provide support for the specification of real-time constraints using real-time temporal logic. Further details of this approach can be found in [Blair99a]. Using this approach, we can verify not only the QoS management subsystem itself, but also its interaction with the rest of the system. The approach to this is based on a *composition algorithm*, which relies on a common underlying semantic model based on *labelled transition systems* [Blair99a]. The overall approach is summarised in figure 2.

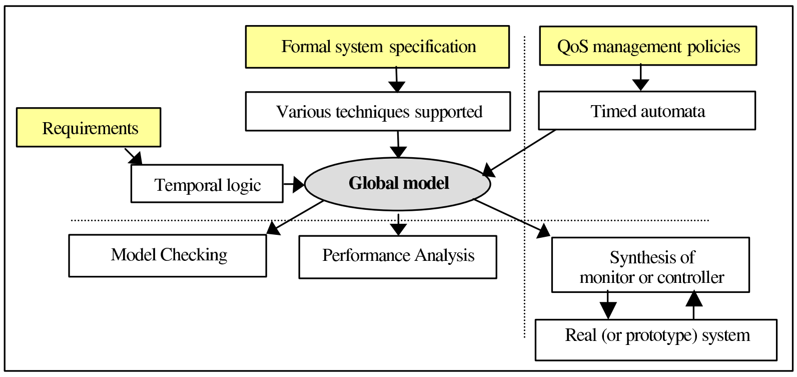

Formal system specification
QoS management policies
Various techniques supported
Timed automata
Requirements
Temporal logic
Global model
Model Checking
Performance Analysis
Synthesis of monitor or controller
Real (or prototype) system

Figure 2: Our multi-paradigm specification approach.

We have also developed a *tool suite* to support this process. Currently, this supports the composition of specifications written in LOTOS, timed automata, and the real-time temporal logic QTL [Blair94]. The output of this process is a combined timed automaton, the behaviour of which can be investigated using either a simulator (to interactively step through behaviour) or a model checker (to verify the desired properties as expressed using temporal logic). In essence, the tool allows us to verify QoS management functions in isolation or in combination with other system behaviour.

Note that timed automata, once verified by the tool, can be used directly as scripts for QoS management functions. To support this, the tool outputs timed automata using a standardised format (FC2 [Madelaine93]). This format is also used as a scripting language for respective QoS management functions. Using this standardised format also encourages interoperability with other formal tools.

5. IMPLEMENTATION

5.1. The Underlying Platform

An implementation of the reflective middleware architecture and associated QoS management functions has been carried out using Python [Watters96]. Python was used for three main reasons. Firstly, the language already has some reflective facilities, on which it has proved to be relatively easy to build our meta-models. Secondly, the language is an ideal vehicle for rapid prototyping, due to its interpreted nature and flexible typing. Thirdly, the language also provides good support for distributed programming. The one drawback with using Python is that, due to its interpreted nature, performance is not comparable to a compiled language. This is however not viewed as critical at this stage of the research. We revisit this argument in section 7 below.

The platform implements the programming model introduced in section 3.1. In more detail, a `Component` class provides one or more interfaces, where an interface can be of type operational, stream or signal. Operational interfaces support operations in the traditional CORBA sense. In contrast, stream interfaces support either the production or consumption of continuous media data. Finally, signal interfaces emit or receive primitive, asynchronous events. Component interfaces in the same address space can be bound together using a `localBind()` primitive. To cross address spaces (or machines), it is necessary however to create a binding component (i.e. explicit binding as discussed in section 3.1).

The `Component` class also supports access to the different meta-models defined in our architecture. At present, the platform focuses mainly on structural reflection with complete implementations of the encapsulation and composition meta-models (although minimal implementations of the other meta-models also exist). The encapsulation meta-model provides operations to inspect the target interfaces (c.f Java introspection), to set, get or delete attributes, to add or delete methods, and to introduce or remove pre-and post-methods. The composition meta-model provides a method to inspect the internal structure of a composite component, returning a representation of the underlying component graph (see section 3.2.1). Operations are then provided to add or remove a component, to create or break a local binding, and to atomically replace one component with another. Access to the meta-spaces is provided by the operations `encapsulation()` and `composition()` respectively (defined on both components and interfaces).

The platform also provides an underlying engineering model heavily influenced by RM-ODP [Blair97]. In particular, the platform provides a `Capsule` class, representing a distinct address space. This class supports operations to create a new component within that address space or to register an existing component with that address space (thus making it accessible from that capsule). Capsules also provide the `localBind()` primitive mentioned above, as well as a corresponding `breakBinding()` operation. In addition, the platform supports a simple name server, together with an extensible set of factories to add new components to the environment (e.g. to create a new binding component). Factories are accessed through the capsule interface as described above.

Crucially, the platform supports the creation of management components for dynamic QoS management (as discussed in section 4 above). Such management components act as interpreters for timed automata (described using FC2), and interact with other objects using signal interfaces. We illustrate the use of the management components with a simple example of a QoS-managed audio stream, which has been implemented on the platform. A second, more detailed example can also be found in [Gancedo99], which also includes a more thorough evaluation of the approach.

5.2. A QoS Management Example

In the example, we have an audio source and sink (on different machines) connected by an audio stream binding component. As part of dynamic QoS management, we would like to monitor the state of the sink's buffer to ensure that it is not "full too often" (FTO) or "empty too often" (ETO). To achieve this, the sink monitor (*snk_monitor*) receives, as input, events *buf_full* and *buf_empty* from the sink's buffer. We assume that an appropriate event collector (which exploits the underlying meta-models to locate and monitor the relevant behaviour) generates these events. The monitor sets a period, P, in which the above events are counted (*nf* and *ne* respectively). If the number of full (resp. empty) events occur more than a pre-determined number of times (N) within period P, then the event *FTO!* (resp. *ETO!*) is output. At the end of the period, all

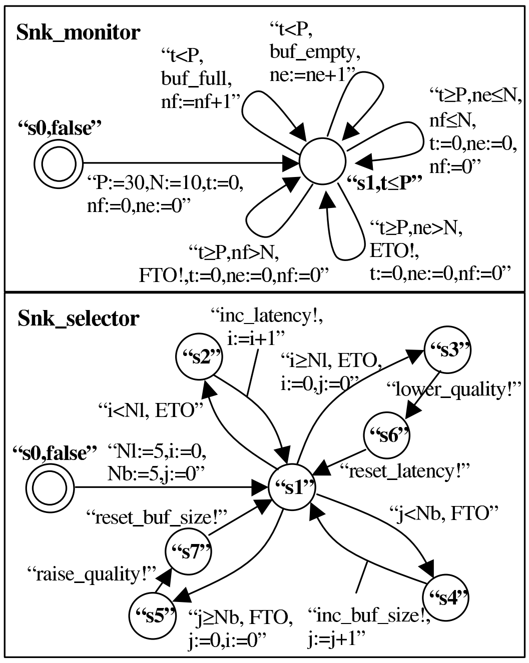


Figure 3: Automata to manage the sink's buffer.

variables are reset.

These two output events are then treated as input events by the sink strategy selector (*snk_selector*). There are various options that the strategy selector can take depending on the sequencing and frequency of the inputs. Firstly, we address the receipt of ETO events, i.e. the sink's buffer is empty too often. We note that the latency of the presentation can be increased to allow the buffer a fixed period of time to (partially) fill up with data (output event *inc_latency!*). However, since we cannot keep increasing the latency indefinitely, we count the number of latency adjustments (i) up to a limit (Nl). Once Nl adjustments have been made, the selector adopts the strategy of asking the source to lower its quality (by emitting an appropriate signal). This strategy also causes the latency to be reset and we try again. If, however, we address the receipt of FTO (full too often) events, the strategy is to adjust the buffer size. Again there is a limit to how many times this should be done. Hence, we count, as above, the number of buffer adjustments (j) up to a limit (Nb). This time, once Nb adjustments have been made, the selector signals to the source to try and raise its quality. This also resets the buffer size. The overall automata are shown in figure 3.

As above, we assume the existence of appropriate strategy activators to bridge between the QoS management subsystem and the underlying functional components. For example, the strategy activator responsible for increasing buffer size operates through the compositional meta-model to locate and re-configure the buffering component.

In this example, it is also possible to combine the monitor and strategy selector into one automaton, using the tool described above (e.g. to improve the efficiency of the QoS management subsystem). This produces a timed automaton with nine states.

Note that this example could also be extended by providing a meta-policy. For example, a meta-manager could monitor the behaviour of the management subsystem and detect the number of events generated by the controller (to change the latency associated with the buffer or the quality of audio sent by the source). If such events are generated too frequently, it is likely that the management subsystem is less than optimal (it could be oscillating unnecessarily between states). Action can then be taken to alter some of the key values associated with the monitor and strategy selector, e.g. N and Nb. A more radical strategy would be to change the strategy selector completely should the current strategy be seen to behave poorly. This is achieved by downloading a new automaton described in FC2.

6. RELATED WORK

6.1. Reflection in Distributed Systems

There is growing interest in the use of reflection in distributed systems. Pioneering work in this general area was carried out by McAffer [McAffer96]. With respect to middleware in particular, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [Singhai97, Román99], focussing on reification of invocation marshalling and dispatching. In addition, researchers at APM have developed an experimental middleware platform called FlexiNet [Hayton97], which allows the programmer to tailor the underlying communications infrastructure by inserting/ removing layers. Their solution is, however, language-specific, i.e. applications must be written in Java. Manola has carried out work in the design of a "RISC" object model for distributed computing [Manola93], i.e. a minimal object model which can be specialised through reflection. Finally, researchers at the Ecole des Mines de Nantes are also investigating the use of reflection in proxy mechanisms for ORBs [Ledoux97].

Our design has been influenced by a number of specific reflective languages. As stated above, the concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [Okamura92]. From this list, it can be seen that AL/1-D does however support a resources model. This resource model supports reification of scheduling and garbage collection of objects (but in a relatively limited way compared to our approach). Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [Watanabe88] and CodA [McAffer96]. Both these systems feature decompositions of meta-space in terms of the acceptance of messages, placing the message in a queue, their selection, and the subsequent dispatching.

Our use of component graphs is inspired by researchers at JAIST in Japan [Hokimoto96]. In their system, adaptation is handled through the use of control scripts written in TCL. Although related to our proposals, the JAIST work does not provide access to the internal details of communication components. Furthermore, the work is not integrated into a middleware platform. The designers of the VuSystem [Lindblad96] and Mash [McCanne97] advocate similar approaches. The same criticisms however also apply to their designs. Microsoft's ActiveX software [Microsoft99] also uses component graphs. This software, however, does not address distribution of component graphs. In addition, the graph is not re-configurable during the presentation of a media stream.

6.2. QoS Management in Distributed Systems

There are many projects investigating QoS management in middleware platforms. BBN's QuO (QUality Objects) project [Vanegas98] is perhaps the closest in approach to our own work in that it adopts a similar open implementation philosophy. QuO, however, adopts an approach reminiscent of aspect-oriented programming [Kiczales97] to specify different aspects of QoS support using a range of specialised (high-level) languages. As such, it provides a more declarative, as opposed to procedural, approach to QoS management.

Similarly, researchers at the University of Illinois have developed a task control model to support dynamic reconfiguration in middleware platforms [Li99]. This project is complementary to the Illinois work on reflective middleware platforms mentioned above, exploiting the openness of the underlying platform for more fine-grained control. Their approach is based on the use of a fuzzy logic inference engine together with a rule base of possible adaptations, in contrast to our use of timed automata.

Other notable research projects include DIMMA [Donaldson98], Tao [Schmidt97], Jonathan [Dumant97], GOPI [Coulson99], and ERDoS [Chatterjee97]. In addition, the OMG have carried out important work in this area with their specifications for the "Control and Management of A/V Streams" [OMG97] and "Realtime CORBA" [OMG98]. However, these projects all tend either to offer only restricted access to underlying middleware functions, or to focus on more static aspects of QoS management (specification, configuration, and resource reservation).

Finally, there is a considerable body of research on specific techniques for QoS management in distributed systems (not necessarily related to middleware platforms). For example, many of the most influential techniques have emerged from Internet applications such as the Mbone tools, e.g. *vic*, [McCanne95]. Other important techniques include the use of filters [Yeadon96], and receiver-driven adaptation through layered encoding [McCanne96]. There is also a growing trend to provide toolkits incorporating such approaches to support the construction of adaptive multimedia applications, e.g. SWiFT [Goel98, Walpole97], Djinn [Mitchell98], and the JAIST research mentioned above [Hokimoto96].

7. CONCLUDING REMARKS

This paper has considered the role of reflection in supporting important QoS management functions. More specifically, the paper has described the design of a novel reflective middleware platform and examined how the dynamic QoS management functions of monitoring and adaptation can be supported in this design. The most important features of the approach are:

i) a procedural approach is adopted, whereby QoS management functions operate on a component graph of underlying components exposed via meta-space;
ii) QoS management is introduced via management components which communicate with other events through event notification (thus providing an important separation of concerns);
iii) three styles of management component are identified, namely monitors, strategy selectors and strategy activators;
iv) monitors and strategy selectors are typically written using a timed automata notation thus encouraging the formal verification of QoS management subsystems.

We believe that the use of reflection provides important benefits over more traditional approaches to QoS management. For example, reflection provides a principled approach to providing access to underlying components. This is crucial in supporting monitoring and adaptation aspects of QoS management. The approach is also dynamic in that management components can be added or removed at any time. In addition, reflection enables the introduction of sophisticated management structures involving both policies and meta-policies.

Finally, the experience from our pilot implementation work (see section 5) provides initial validation for our claims. Although performance is not a primary concern at this stage, the platform was able to deliver the required real-time performance for the audio stream example. In the future, we plan to carry out further implementation experiments and to extend the work by considering more sophisticated examples of QoS management policies and indeed meta-policies. Ongoing studies are also focussing on the issue of performance. In particular, we are currently implementing a lightweight, reflective component model, based on COM, offering access to each of the meta-space models. A first implementation of this component model has now been completed. We are now using the reflective components to construct a configurable and re-configurable CORBA-base platform following the architecture described above. The ultimate aim of this work is to demonstrate that flexibility is not necessarily at the expense of performance.

ACKNOWLEDGEMENTS

The research described in this paper is partly funded by CNET, France Telecom (CNET Grant 96-1B-239). Particular thanks are due to Jean-Bernard Stefani and his group at CNET for many useful discussions on reflection and distributed systems. The work on specification and formal verification of QoS properties is funded by EPSRC (Research Grant GR/L28890). We would also like to acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK, and the Universities of Oslo and Tromsø (all in Norway). Particular thanks to Frank Eliassen, Vera Goebel, Oyvind Hanssen and Thomas Plagemann. Finally, we would like to acknowledge the contributions of a number of researchers at Lancaster to the ideas described in this paper, namely Mike Clarke, Fabio Costa, Tom Fitzpatrick, Hector Duran, Nikos Parlavantzas, Philippe Robin and Katia Saikoski.

REFERENCES

[Agha91] Agha, G., "The Structure and Semantics of Actor Languages", Lecture Notes in Computer Science, Vol. 489, pp 1-59, Springer, 1991.

[Blair94] Blair, L., "The Formal Specification and Verification of Distributed Multimedia Systems", PhD thesis, Available from the Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, 1994.

[Blair97] Blair, G.S., J.B. Stefani, "Open Distributed Processing and Multimedia", Addison-Wesley, 1997.

[Blair98] Blair, G.S., G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, 1998.

[Blair99a] Blair, L., G.S. Blair, "Composition in Multi-paradigm Specification Techniques", In Proceedings of the 3rd International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'99), February 15th-18th, Florence, Italy, P. Ciancarini, A. Fantechi, R. Gorrieri (eds), Kluwer, 1999.

[Blair99b] Blair, G.S., F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, J.B. Stefani, "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp115-134, 1999.

[Chatterjee97] S. Chatterjee, J. Sydir and B. Sabata, "Modeling Applications for Adaptive QoS-based Resource Management", Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop, pp 194-201, Bethesda, Maryland, August 1997.

[Costa98] Costa, F., G.S. Blair, G. Coulson, "Experiments with Reflective Middleware", Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98 Workshop Reader, Springer-Verlag, 1998.

[Coulson99] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.

[Donaldson98] Donaldson, D., Faupel, M., Hayton, R., Herbert, A., Howarth, N., Kramer, A., MacMillan, I., Otway D. and Waterhouse, S., "DIMMA - A Multi-media ORB", Proc. Middleware '98, The Low Wood Hotel, Ambleside, England, September 1998.

[Dumant97] Dumant, B., F. Horn, F. Dang Tran, J.B. Stefani, "Jonathan: An Open Distributed Processing Environment in Java", Proc. IFIP International

Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Fitzpatrick98] Fitzpatrick, T., G.S. Blair, G. Coulson, N. Davies, P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings", Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE, 1998.

[Gancedo99] Gancedo, D.S., "QoSMonAuTA: QoS Monitoring and Adaptation using Timed Automata", MSc Thesis, Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, September 1999.

[Goel98] Goel, A., D. Steere, C. Pu, and J. Walpole, "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit", Technical Report 98-009, Computer Science and Engineering Dept., Oregon Graduate Institute of Science and Technology, USA, 1999.

[Hayton97] Hayton, R., "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, 1997.

[Hokimoto96] Hokimoto, A, T. Nakajima, "An Approach for Constructing Mobile Applications using Service Proxies", Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96), IEEE, 1996.

[Kiczales91] Kiczales, G., J. des Rivières, D.G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.

[Kiczales97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, J. Irwin, "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag, June 1997.

[Román99] Román, M., F. Kon, R. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case", Proceedings of the ICDCS'99 Workshop on Middleware, Austin, Texas, May-June 1999.

[Ledoux97] Ledoux, T., "Implementing Proxy Objects in a Reflective ORB", Proc. ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, 1997.

[Li99] Li, B. and K. Nahrstedt, "Dynamic Reconfiguration for Complex Multimedia Applications,Proceedings of the IEEE International Conference on Multimedia Computing and Systems (IEEE Multimedia Systems '99), Florence, Italy, June 7-11, 1999.

[Lindblad96] Lindblad, C.J., D.L. Tennenhouse, "The VuSystem: A Programming System for Computer-Intensive Multimedia", Journal of Selected Areas in Communications, Vol. 14, No. 7, pp 1298-1313, IEEE, 1996.

[Madelaine93] E. Madelaine, R. de Simone, "FC2: Reference Manual Version 1.1", see http://www.inria.fr/meije/verification/doc.html, 1994.

[Manola93] Manola, F., "MetaObject Protocol Concepts for a "RISC" Object Model", Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, 1993.

[McAffer96] McAffer, J., "Meta-Level Architecture Support for Distributed Objects", In Proceedings of Reflection 96, G. Kiczales (end), pp 39-62, San Francisco; Also available from Department of Information Science, The University of Tokyo, 1996.

[McCanne95] McCanne, S. and Jacobson, V., "vic: A Flexible Framework for Packet Video, Proc. ACM Multimedia '95, San Francisco, USA, pp 511-522, November 1995.

[McCanne96] McCanne, S. and Jacobson, V., "Receiver Driven Layered Multicast", Proc. ACM SIGCOMM'96, pp 117-130, Stanford, CA, USA, 1996.

[McCanne97] McCanne, S., E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T-K. Tung, D. Wu, "Towards a Common Infrastructure for Multimedia-Networking Middleware", Proc. 7th International Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St Louis, Missouri, 1997.

[Microsoft99] Microsoft ActiveX Home Page: http://www.microsoft.com/com/tech/activex.asp.

[Mitchell98] Mitchell, S., H. Naguib, G. Coulouris, and T. Kindberg, "A QoS Support Framework for Dynamically Reconfigurable Multimedia Applications", Proc. DAIS'99, Helsinki, Finland, June 1999.

[Okamura92] Okamura, H., Y. Ishikawa, M. Tokoro, "AL-1/d: A Distributed Programming System with

Multi-Model Reflection Framework", Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[OMG97] Control and Management of Audio Visual Streams, OMG Document number telecom/97-05-07; available from http://www.omg.org/library/schedule/AV_Streams_RTF.htm

[OMG98] Realtime CORBA V1.1, Initial Submission to Realtime SIG's RFP on Realtime CORBA, OMG Document number orbos/98-01-08.

[OMG99] The Common Object Request Broker: Architecture and Specification versions 3.0., available at http://www.omg.org/.

[Saikoski99] Saikoski, K.B. and Coulson, G., "Adaptive Groups in OpenORB", Proceedings of the 6th Doctoral Consortium on Advanced Information System Engineering (CAiSE'99 DC), Heidelberg, Germany, 14-16 June 1999.

[Schmidt97] Schmidt, D.C., R. Bector, D. Levine, S. Mungee, G. Parulkar, "An ORB End System Architecture for Statically Scheduled Real-Time Applications", Proceedings of the Workshop on Middleware for Real-Time Systems and Services", IEEE, San Francisco, 1997.

[Singhai97] Singhai, A., R. Campbell, "Reflective ORBs: Supporting Robust, Time-critical Distribution", Proc. ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems, Jyväskylä, Finland, 1997.

[Smith82] Smith, B.C., "Procedural Reflection in Programming Languages", PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982.

[Szyperski98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Vanegas98] Vanegas, R., J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Walpole97] Walpole, J., R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu, "A Player for Adaptive MPEG Video Streaming Over The Internet", Proc. 26th AIPR-97, SPIE, Washington DC, October 15-17, 1997.

[Watanabe88] Watanabe, T., A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language", In Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of "Object-Oriented Concurrent Programming", A. Yonezawa, M. Tokoro (eds.), pp 45-70, MIT Press, 1987.

[Watters96] Watters, A., G. van Rossum, J. Ahlstrom, "Internet Programming with Python", Henry Holt (MIS/M&T Books), 1996.

[Yeadon96] Yeadon, N., Garcia, F., Shepherd, D., and Hutchison, D., "Filters: QoS Support Mechanisms for Multipeer Communications, IEEE Journal on Selected Areas in Communications, Special Issue on Distributed Multimedia Systems and Technology, Vol. 14, No. 7, pp 1245-1262, September 1996.

[Yokote92] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation", In Proceedings of OOPSLA'92, ACM SIGPLAN Notices, Vol. 28, pp 414-434, ACM Press, 1992.

Notes

1. This journal is available online at http://ioj.iee.org.uk/journals/ip-sen/. Volume 147, Issue 1 is available at http://ioj.iee.org.uk/journals/ip-sen/2000/01/. More information is available from http://www.iee.org.uk/publish/journals/

“OOPP, A Reflective Middleware Platform including Quality of Service Management”. A dissertation submitted in partial fulfillment for the degree of Doctor Scientiarum at Department of Computer Science, Faculty of Science, University of Tromsø. Anders Andersen, February 2, 2002.

ISBN 82-7747-097-5
NORUT IT Report IT302/1-99