Building a CORBA Gateway for Rational RoseRT

by

Li Zhao

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Science

in

Information and Systems Science

School of Computer Science

Carleton University

Ottawa, Ontario

27 January 2004

© Copyright

2004, Li Zhao

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89878-4

Our file Notre référence

ISBN: 0-612-89878-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

The Object Management Group (OMG)'s Model Driven Architecture (MDA) approach tries to separate the design of an application's logic into the details pertaining to the communication platform, deployment, the OS and hardware architecture. The designer is encouraged to design a system with a platform independent modeling language and the platform dependent model is generated by tools which implements the standard mapping between the PIM and PSM.

The goal of this thesis is to provide RoseRT applications with the ability to communicate with remote CORBA applications with minimal impact to the original RoseRT model.

This thesis first extends the OMG's UML Profile for CORBA in defining a RoseRT profile for CORBA, enabling mapping between a CORBA IDL interface and RoseRT model elements. A CORBAProxy capsule is then designed and implemented to facilitate the application capsule to participate in CORBA communications, taking the mapping as given. A case study is used to illustrate both the mapping, and the implementation of the CORBAProxy.

Acknowledgements

First I wish to thank my supervisor Dr. Francis Bordeleau for guiding me in the research and writing of this thesis. I also want to thank Mr. John Hogg from IBM for helping me in issues related in Rose RealTime and Model Driven Architecture. Discussions with Mr. Juan Pablo Zamora, PhD. candidate at School of Computer Science also help the clarification of mind. Last but not least, I would thank my husband for his support and patience during the whole time.

Table of Contents

CHAPTER 1	INTRODUCTION.....	1
1.1	CONTEXT AND MOTIVATION.....	1
1.1.1	<i>RoseRT</i>	1
1.1.2	<i>CORBA</i>	2
1.1.3	<i>MDA</i>	2
1.2	PROBLEM.....	3
1.3	OBJECTIVES.....	3
1.4	CONTRIBUTIONS.....	4
1.5	OUTLINE OF THE THESIS.....	5
CHAPTER 2	BACKGROUND.....	6
2.1	MDA.....	6
2.1.1	<i>MDA and the Software Life Cycle</i>	6
2.1.2	<i>Model Mappings</i>	8
2.2	ROSERT.....	9
2.2.1	<i>RoseRT Modeling Constructs</i>	10
2.2.2	<i>Distribution of RoseRT Applications through Connexis®</i>	17
2.3	CORBA.....	20
2.3.1	<i>CORBA Object Model Concepts and Definitions</i>	20
2.3.2	<i>CORBA Interface Definition Language (IDL)</i>	25
2.3.3	<i>CORBA Request Flow</i>	26
2.4	SUMMARY.....	28
CHAPTER 3	ROSERT PROFILE FOR CORBA.....	29
3.1	INTRODUCTION.....	29
3.2	CORBA STRUCT.....	31
3.2.1	<i>RoseRT Model Elements</i>	31
3.2.2	<i>Notation</i>	32
3.2.3	<i>Differences from UML Profile for CORBA v1.0</i>	32
3.3	CORBA IDL INTERFACE.....	32
3.3.1	<i>RoseRT Model Elements</i>	32
3.3.2	<i>Notation</i>	33
3.3.3	<i>Differences from UML Profile for CORBA v1.0</i>	34
3.3	CORBA IDL ATTRIBUTES.....	35
3.3.1	<i>RoseRT Model Elements</i>	35
3.3.2	<i>Notation</i>	36
3.3.3	<i>Differences from UML Profile for CORBA v1.0</i>	37
3.4	CORBA STRUCT DERIVED FROM CORBA OPERATIONS.....	37
3.4.1	<i>RoseRT Model Elements</i>	37
3.4.2	<i>Notation</i>	39
3.4.3	<i>Differences from UML Profile for CORBA v1.0</i>	40
3.5	CORBA OPERATIONS.....	40
3.5.1	<i>RoseRT Model Elements</i>	40

3.5.2	<i>Notation.....</i>	41
3.5.3	<i>Differences from UML Profile for CORBA v1.0.....</i>	41
3.6	SUMMARY.....	42
CHAPTER 4 REQUIREMENT ANALYSIS OF THE CORBA PROXY		43
4.1	THE CORBAPROXY IN DISTRIBUTED ROSEXT COMMUNICATION.....	43
4.2	USE CASES FOR THE CORBAPROXY	45
4.2.1	<i>Use Case Diagram.....</i>	45
4.2.2	<i>External Actors</i>	45
4.2.3	<i>Use Case Descriptions.....</i>	47
4.3	CONCEPT DESCRIPTIONS.....	52
CHAPTER 5 IMPLEMENTATIONS OF THE CORBAPROXY		55
5.1	CASE STUDY: RESTAURANT FOOD ORDERING SYSTEM	55
5.1.1	<i>Overview</i>	55
5.1.2	<i>Description of the Restaurant Food Ordering System (RFOS)</i>	56
5.1.3	<i>The Complete IDL Specification.....</i>	58
5.2	COMPONENTS OF CORBAPROXY	60
5.2.1	<i>Servants.....</i>	60
5.2.2	<i>Capsules.....</i>	62
5.2.3	<i>Protocols.....</i>	65
5.3	IMPLEMENTATION OF THE USE CASES.....	66
5.3.1	<i>Publish Service for the AppCapsule</i>	66
5.3.2	<i>Process Request from CORBA Client.....</i>	71
5.3.3	<i>Obtain Reference of Remote CORBA Object.....</i>	76
5.3.4	<i>Send out CORBA Request to Remote CORBA Object</i>	80
5.4	CONFIGURE A ROSEXT APPLICATION FOR DISTRIBUTION	85
5.4.1	<i>Generate the CORBAProxy</i>	86
5.4.2	<i>Structural changes</i>	89
5.4.3	<i>Behavioral changes.....</i>	90
5.5	TESTING OF THE CORBAPROXY	90
CHAPTER 6 CONCLUSION AND FUTURE WORK.....		90
6.1	CONCLUSION.....	90
6.2	FUTURE WORK	92

List of Figures

Figure 1 Example of MDA development approach for a CORBA application.....	8
Figure 2 Capsule Diagram	12
Figure 3 Example of a RoseRT Protocol	13
Figure 4 Capsule and Ports	14
Figure 5 Capsule Containment.....	15
Figure 6 Capsule Structure Diagram.....	15
Figure 7 State Diagram of the Hand Held Device	16
Figure 8 Ping Pong with Binding on Wired Ports	17
Figure 9 Structure and State Diagram of Capsule Pong	18
Figure 10 Ping and Pong Communicate through Unwired Port	18
Figure 11 Pong's Structure and State Diagram after Configure for Connexis.....	19
Figure 12 CORBA Request Flow	27
Figure 13 MDA Context of the RoseRT to CORBA Mapping	29
Figure 14 UML Profiles as Ways of Defining Model Mappings	30
Figure 15 Virtual Metamodel for CORBA Struct.....	32
Figure 16 Struct Example	32
Figure 17 Virtual Metamodel for CORBA Interface.....	33
Figure 18 Interface Example.....	34
Figure 19 Virtual Metamodel for CORBA Attribute.....	36
Figure 20 Attribute Example	37
Figure 21 Virtual Metamodel for CORBA Struct Derived from CORBA Operation	39
Figure 22 CORBA Derived Struct Example.....	40
Figure 23 Virtual Metamodel for CORBA Operation	41
Figure 24 Operation Example.....	41
Figure 25 Architecture of Distributed RoseRT Application.....	43
Figure 26 Use Case Diagram	45
Figure 27 Concepts of the CORBA-distributed RoseRT Application.....	52
Figure 28 RoseRT Data Types Correspond to CORBA Data Types Defined in the IDL	59
Figure 29 RoseRT Protocols Correspond to CORBA Interfaces Defined in IDL.....	59
Figure 30 Servant as the Bridge between CORBA Operation and RoseRT Signal.....	60
Figure 31 Structure of the Restaurant System	62
Figure 32 Internal Structure of the CORBAProxy	63
Figure 33 Structure of ORBContainer	63
Figure 34 Class Diagram -- Publish Service.....	67
Figure 35 Sequence Diagram -- Publish Service	68
Figure 36 Class Diagram -- Process Request	72
Figure 37 Sequence Diagram -- Process Request.....	74
Figure 38 Class Diagram -- Obtain Reference.....	77
Figure 39 Sequence Diagram -- Obtain Object Reference	78
Figure 40 BoundServiceList	79
Figure 41 Class Diagram -- Send CORBA Request.....	80
Figure 42 Sequence Diagram -- Send CORBA Request	81

Chapter 1 Introduction

In this chapter, we briefly introduce the context of the thesis in Section 1.1, and then state the problem and the objectives of the thesis in Section 1.2 and 1.3. Section 1.4 outlines the main contribution of this work and Section 1.5 summarizes the structure of the thesis.

1.1 Context and Motivation

Several concepts are essential to the work of this thesis. Foremost is IBM Rational Rose® RealTime, whose applications we attempt to integrate with a distributed communication middleware. The second component is CORBA, the middleware technology we selected for integration. Finally, we use the MDA initiative proposed by OMG as the methodology for the proposed solution in this thesis.

1.1.1 RoseRT

RoseRT (IBM Rational Rose® RealTime[18]) is a software development environment that allows developers to create models of the software system based on the Unified Modeling Language (UML) constructs, to generate implementation code, and then compile, run and debug the application. The executable model employed in RoseRT specializes in representing event-driven, concurrent applications.

Currently, communication between a RoseRT application and another distributed application is optimized only when the other application is also a RoseRT application. The connectivity between two RoseRT applications is provided by Rational Connexis, a Rational RoseRT add-in product. If one party to the communication is not a RoseRT application, developers have to implement a special IPC mechanism for the RoseRT

application for each different type of application that the RoseRT application needs to talk to.

If, instead of communicating with distributed applications directly, we develop the RoseRT application on top of a communication middleware, the RoseRT application will be able to deal with different applications in a homogeneous way.

1.1.2 CORBA

CORBA (Common Object Request Broker Architecture) provides platform-independent programming interfaces and models for portable distributed object-oriented computing applications. It is independent of programming languages, computing platforms, and networking protocols.

CORBA specifications, written and maintained by Object Management Group (OMG), supply a set of flexible abstractions and concrete services needed for communication within a heterogeneous computing environment.

1.1.3 MDA

Model Driven Architecture is an OMG initiative to better address the problem of integration during software system development. The basic idea of MDA is to use UML to specify both the static interfaces and the dynamic behavior of components in a platform independent model (PIM). To define platform specific models (PSMs), the MDA specifications define a number of mappings from PSMs to UML profiles, which are platform specific extensions of UML.

In the short term, MDA allows designers' UML models to be translated into code. Over the long term, MDA makes changing platforms, OS and hardware easier by

mapping the same PIM into the new PSM and then generating new code for the new platforms.

As an extension to UML, RoseRT is able to express both platform independent and platform specific models. Currently the integration of RoseRT applications with distribution middleware is limited to Rational Connexis, which allows distributed communication only among RoseRT applications. CORBA, on the other hand, provides for distributed communication among heterogeneous applications. Using CORBA as the communication middleware will provide more flexibility for the RoseRT application.

1.2 Problem

This thesis focuses on the problem of integrating communication middleware into RoseRT. The middleware technology considered here is CORBA. The main problems are to transform CORBA requests into RoseRT signals and CORBA message data into RoseRT data.

1.3 Objectives

The main objective of this thesis is to effect smooth integration between RoseRT applications and an Object Request Broker (ORB), thus enabling RoseRT applications to communicate with remote CORBA applications. The process of integration should satisfy the following requirements:

- The timing of making the decision to integrate should be flexible; it could happen before, during or after the implementation of RoseRT applications.
- The integration process should have minimal impact on the structure and behavior of the RoseRT application.

- The RoseRT application should be able to communicate with distributed CORBA applications as if it were communicating with another RoseRT application.

1.4 Contributions

The task of this thesis is to realize the distribution of a RoseRT model using CORBA as the communication middleware.

Following the lead of the MDA approach, we separate this task into the following steps:

1. Define a mapping between the metamodels of RoseRT and CORBA.

The thesis defines a “RoseRT Profile for CORBA” by adopting the format from OMG’s UML Profile for CORBA. The UML Profile for CORBA specifications was designed to provide a standard means for expressing the semantics of CORBA IDL using UML notation and thus to support expressing these semantics with UML tools. The RoseRT Profile for CORBA establishes one to one correspondence between elements in the RoseRT metamodel and elements of the CORBA metamodel. RoseRT is itself an extension to standard UML. This profile for CORBA differs from UML’s when the same CORBA model element is mapped into non-UML defined RoseRT model elements. We omit the mappings that are the same as those defined in the UML Profile.

2. Design a component that utilizes this mapping and enables a RoseRT application to communicate with a remote CORBA application.

We first identify the responsibilities of such a component, which is called the CORBAProxy throughout this thesis. We also introduce all the parties that will interact

with the CORBAProxy during a CORBA communication. Some parties are dictated by the mapping we define in Chapter 3, while others are required by the CORBA communication.

3. Implement the component.

The CORBAProxy is implemented as a RoseRT capsule. Using the mapping defined in the RoseRT Profile for CORBA, it transforms CORBA requests into RoseRT signals and vice versa, allowing the RoseRT application to communicate with remote CORBA applications by sending and receiving RoseRT signals.

1.5 Outline of the Thesis

Following Chapter 1's introduction to the context of this thesis, Chapter 2 briefly introduces the concept of MDA and the two modeling paradigms—IBM Rational Rose® RealTime model and CORBA's Object Model—between which we try to provide a bridge. Chapter 3 defines the mapping between CORBA model elements and RoseRT model elements in terms of the "RoseRT Profile for CORBA." Chapter 4 proposes a component called CORBAProxy as the bridge between a RoseRT application and the ORB. This chapter contains the requirement analysis of this component. Chapter 5 explains in detail our implementation of the CORBAProxy. Finally, Chapter 6 discusses the limitations of our implementation and proposes some alternative solutions, which are more elegant but also more complex given our time constraints.

Chapter 2 Background

This chapter provides the background information for this thesis. Section 2.1 first introduces the MDA initiative proposed by the OMG and its implications for our work in this thesis. Before we demonstrate how to build a bridge between RoseRT applications and the CORBA middleware, we need to develop a better understanding of their individual characteristics. Section 2.2 familiarizes the reader with some important model elements in RoseRT. The differences between RoseRT and UML are also discussed there. Section 2.3 introduces the CORBA object model, including the components the model consists of and how they collaborate with each other during CORBA communications. We also introduce informally the syntax of the CORBA IDL, using a sample IDL excerpted from our case study. The UML Profile for CORBA is discussed briefly in this section.

2.1 MDA

This section explains the OMG's MDA approach to the software development process in more detail. Section 2.1.1 describes the role that Model Driven Architecture plays during the lifecycle of software systems. A simple example is given to illustrate the basic steps taken in a model driven development process. Section 2.1.2 emphasizes the importance of model mapping in the MDA. UML profiles are introduced in this Section as a way of defining mappings between PSMs and extended UML models.

2.1.1 MDA and the Software Life Cycle

Model Driven Architecture [8] (MDA) is an OMG initiative to solve the problem of integration during the process of software development. Historically, during the

development process, application logic has often been mixed with the initialization, configuration and activation of different middleware components, making it tightly coupled with a specific platform. Changing any platform technology later may require rewriting the application code.

MDA supports many of the commonly used steps in model driven component based development and deployment. A key aspect of MDA is that it addresses the complete life cycle of the analysis, design and programming aspects (testing and component building or component assembly), as well as the deployment and management aspects.

The basic idea of MDA is to use UML to fully specify both the static interfaces and the dynamic behavior of components in a platform independent model (PIM). To define platform specific models (PSMs), the MDA specifications define a number of mappings from PSMs to UML profiles, which are platform specific extensions of UML. In the short term, MDA allows designers' UML models to be translated into code. Over the long term, MDA makes changing of platforms, OS and hardware easier. The same PIMs expressed in UML can be generated into new PSM expressed in UML profiles according to PIM to PSM mapping. The new PSM is then generated into new codes.

The OMG has adopted a number of these UML profiles, such as a UML profile for CORBA and a UML profile for Enterprise Distributed Object Computing (EDOC), and is working on further profiles, such as UML profile for Schedulability, Performance and Time.

Figure 1 Example of MDA development approach for a CORBA application

Figure 1 shows the artifacts and their dependencies involved in using the MDA approach to develop distributed applications for the CORBA platform. The business logic of the application is specified in a PIM by a standard UML tool that stores the model in a particular format (possibly in XMI format). A mapping tool that implements the OMG standard mapping of UML to CORBA then generates a PSM that uses UML extensions defined in the UML profile for CORBA. Another tool can then translate the PSM into the different artifacts that are necessary for the deployment of a CORBA application, such as the source code in the programming language of choice and a makefile that is needed to compile the code

2.1.2 Model Mappings

The proposed application development process of MDA requires that there exist mappings to relate models of the same system at different level of abstractions. It is essential that these mappings are defined precisely and automated as far as possible. For

example, in order to automatically generate code from a design model, we need mappings between a PIM and a PSM (such as CORBA), and then to the programming language code of developer's choice.

The definition of profiles is enabled by UML's extension mechanisms. A UML profile is defined in terms of stereotypes and tagged values. The CORBA profile, for example, defines a stereotype `<<CORBAInterface>>` that indicates that a UML class models an interface defined in a CORBA IDL.

Local interfaces are represented using the TaggedValue { isLocal = TRUE }. The profiles also list a number of consistency constraints that extend the consistency constraints defined in the UML semantics guides. For example, the UML profile for CORBA adds the following constraints to a UML Class that has the stereotype of `<<CORBAInterface>>`: all attributes of a `<<CORBAInterface>>`-stereotyped Class must have visibility 'public', all navigable opposite AssociationEnds of a `<<CORBAInterface>>`-stereotyped Class must have visibility 'public', and so on.

2.2 RoseRT

RoseRT is a software development concept and notation environment that enables developers to create models of the software system based on Unified Modeling Language

constructs. It allows users to build executable models and provides different levels of testing.

RoseRT can be used during all phases of the software development lifecycle, from requirements analysis through design, implementation, testing and final deployment. It extends UML to support real-time application. It also integrates with other tools required during the different phases of development. For example, developers work directly at the modeling level through RoseRT to generate both the architecture—using the capsule diagram—and the behavior—using the state diagram—of the system under development. Furthermore, they can make use of detailed programming language code on the state machine’s transition or the definition of a class’s operation to further enhance the behavior of the system. A scripting language tool can take both the model and the detailed code and generate source files in the same programming language for compilation. The actual compilation is performed by a compiler/linker outside of the toolset.

In this section, we first familiarize the reader with the RoseRT concepts and notation in Section 2.2.1, including model elements and their graphic representation. Section 2.2.2 discusses the proprietary distribution middleware for RoseRT applications— Rational Connexis. We use an example to illustrate the procedure used to integrate the Connexis middleware into a RoseRT application.

2.2.1 RoseRT Modeling Constructs

In addition to supporting the core UML constructs, RoseRT uses the extensibility features of UML to define some new constructs that are specialized for real-time system

development. These new constructs allow code generation of elements that can use the services provided in the Services Library, such as state machines, concurrency, message passing, and timing services. A brief introduction to those constructs follows.

A restaurant food ordering system example is used throughout this Section to illustrate the use of each RoseRT modeling element. In this restaurant, waiters take orders in the dining area with handheld devices (*HandHeldDevice*). The orders are then sent by the handheld device to the control terminal located in the kitchen (*KitchenTerminal*). Once the food is ready, the *KitchenTerminal* will signal the appropriate *HandHeldDevice*, notifying the waiter to pick up and serve the dishes. A detailed description of this case study can be found in Section 5.1.

2.2.1.1 Classes and Capsules

The definition of Class used in RoseRT is the same as that defined in UML: a classifier that describes a set of objects that share the same specifications of features, constraints, and semantics [6]. To better model concurrent objects, RoseRT uses the extensibility features of UML to define the concept of Capsules based on Classes. Capsules have many of the same properties as classes; for example, they can have operations and attributes. Capsules may also participate in dependency, generalization, and association relationships. However, they also have several specialized properties that distinguish them from classes. Their semantics are similar to the “Active Class”

metaclass defined in UML.¹ Classes as used in RoseRT are passive classes that represent data types used for storage or passed by messages exchanged between capsules.

Figure 2 Capsule Diagram

Figure 2 shows the graphic notation for the capsule. It is the same as a class except for the symbol in the upper-right corner and the stereotype <<Capsule>> shown above the identifier of the capsule.

Conceptually, a capsule has its own thread of control in a system. Capsules are highly encapsulated. They can only communicate with other capsules by sending messages through their public ports (see Section 2.2.1.2). The advantage of message-based interfaces is that a capsule has no knowledge of its environment outside of these interfaces, making it much more distributable, reusable, and robust than regular objects.

There are two diagrams associated with each capsule: a capsule structure diagram (Section 2.2.1.3) and a state diagram (Section 2.2.1.4), which specify the internal structure and behavior of a capsule.

2.2.1.2 Protocols and Ports

A capsule sends and receives messages through its ports. Ports are owned by the capsule instance in the sense that they are created along with their capsule and destroyed

¹ An active class is a class whose instances may execute their own behavior without requiring method invocation. This is sometimes referred to as “the object having its own thread of control.” The points at which an active object responds to communications from other objects are determined solely by the behavior of the active object and not by the invoking object. This implies that an active object is both autonomous and interactive to some degree [6].

when the capsule is destroyed. The ports are in turn connected to other capsules, enabling the transmission of messages among capsules. The messages that can be sent out or received from a port are defined by protocols. A port is an instance of a protocol. The signals that a port is able to send and receive are defined by its protocol.

Figure 3 shows a protocol defining the set of messages passing between kitchen terminal and handheld device.

Figure 3 Example of a RoseRT Protocol

As shown in the class diagram, a RoseRT protocol contains a symbol and has the stereotype <<Protocol>>. When defining a protocol, the developer needs to specify the In Signals and Out Signals. These signals are also shown on the diagram, with denoting incoming and outgoing signals. Messages sent between capsules contain a required signal name (which identifies the message), an optional priority (the relative importance of this message compared with other unprocessed messages on the same thread), and optional application data.

Notice that the direction of signals used by the receiver and the sender must be opposite. There are therefore different perspectives of a protocol, which are called *protocol roles*. Protocol roles represent the communication from the perspective of one participant in the communication scenario. The capsule that contains the *base protocol*

role will then be able to receive messages defined in “In Signals” and send messages defined in “Out Signals”. The capsule that contains the *conjugate protocol role* will be able to send the “In Signals” and receive the “Out Signals”. Any port realizes either one of the two protocol roles.

Figure 4 Capsule and Ports

The class diagram on the left of Figure 4 shows the aggregation relationship between the HandHeldDeviceCap and the OrderProtocol. The capsule structure diagram of the former shows the port that implements the base role of the latter.

2.2.1.3 Capsule Structure Diagrams

As a specialized class, the capsule’s structure is defined by its attributes. Unlike an ordinary UML class, capsules can have attributes that are ports or other capsules (called *capsule roles*). Capsule structure diagrams have been introduced to specify the capsule's interface (public ports) and its internal composition (capsule roles and protected ports). This diagram also allows one to specify the communication relationships between capsules. Below is a simple example of the capsule structure diagram.

Figure 5 Capsule Containment

Figure 6 Capsule Structure Diagram

Figure 5 shows, via the class diagram, in terms of class diagram that the capsule *RestaurantSystem* aggregates two other capsules: *KitchenTerminalCap* and *HandHeldDeviceCap*. Figure 6 shows the capsule structure diagram of the capsule *RestaurantSystem*. There are two capsule roles: *handHeldDevice* and *kitchenTerminal*. Notice that both capsule roles contain a port of the protocol *OrderProtocol* but the two protocol roles are conjugate with each other, with one showing in black and the other white. The link between the two ports is the binding that enables the *handHeldDevice* to send and receive messages from the *kitchenTerminal*. Defining links between ports of

different capsule roles in this way, we can enable the communication of capsules that reside within the same process. With the help of Connexis, we can also achieve communication between capsule roles residing in different processes in a similar way.

2.2.1.4 State Diagrams

RoseRT's state diagrams consist of a subset of the state machine notation defined in the UML.² A typical state diagram contains an initial pseudo state, states and transitions. Users can model the behavior of a capsule by creating states of the capsule and linking the states with transitions. Transitions are triggered by a port and signal pair. RoseRT allows language specific code to be put in the state machine, as either the actions taken when entering or exiting a state or the activities performed during the transition from one state to the next. A capsule's behavior is defined by a state machine that is driven by the receipt of messages through the capsule's ports.

Figure 7 State Diagram of the Hand Held Device

² RoseRT's state machine is actually a subset of UML 2.0's state machine. It differs more substantially from the state machine defined in UML 1.x. Since the concept of state machine is not essential to the work of this thesis, these differences are not discussed in further details.

Figure 7 shows the state diagram of the HandHeldDeviceCap. Upon initialization, the handheld device is in ready state. When a customer needs to place a new order, it enters into the *placing order* state, as new dishes are entered or edited. After the customer finishes ordering, the handheld device goes back to the ready state, waiting for the next request. Detailed activities can be coded into the transition. For example, during the transition *finishOrder*, the handheld device could send the newly taken order to the kitchen terminal, notifying the cooks to prepare the food.

2.2.2 Distribution of RoseRT Applications through Connexis®

Connexis is a distribution component developed by Rational Software that provides connectivity for RoseRT applications. Connexis is tightly integrated with RoseRT and is highly optimized for event-driven asynchronous systems [18].

In this section, we will take a simple Ping-Pong example from the Connexis Users Guide [18] to see how RoseRT executables can be distributed using Connexis. This material is later compared with the process involved in integrating with CORBA using the component designed and implemented by this thesis.

Figure 8 Ping Pong with Binding on Wired Ports

Figure 9 Structure and State Diagram of Capsule Pong

The two capsules Ping and Pong both contain ports that implement opposite protocol roles of the same protocol. When both capsules run within the same process, their communication link is modeled by the binding between their wired ports, as shown in Figure 8. Figure 9 shows the internal structure and state machine for the capsule Pong.

Capsule Ping has a similar structure and states. Notice that the symbol is used in RoseRT to represent a wired port. Wired ports are explicitly connected to other wired ports with connectors, but unwired ports are not connected during design. Instead, they are dynamically connected at run-time[18].

Figure 10 Ping and Pong Communicate through Unwired Port

Figure 11 Pong's Structure and State Diagram after Configure for Connexis

When Ping and Pong do not reside in the same process, the connector between the two wired ports is no longer suitable for modeling the communication link between the two capsules. To support distributed RoseRT applications using Connexis, we need to do the following:

- Remove the connector between the two wired ports (Figure 10).
- Change the port of the capsule from wired to unwired, the latter denoted by the symbol .
- Configure capsules Ping and Pong for Connexis (Figure 11), so that they contain a Connexis specific capsule role to handle inter-process communication for them.
- Modify the state machine of the capsule to wait for Connexis' Distributed Connection Service (DCS) to start before sending and receiving any messages through the unwired ports (Figure 11).

Connexis also provides a naming service to allow the application to register unwired ports using a specially formatted string. Other distributed RoseRT models could bind to registered ports through this service string and irrespective of the actual location of the executables the port is run in. The AppCapsule uses the unwired port in the same

way it uses the wired port. The signal, however, will be handled by Connexis DCS before it is transported through the network and reaches the Connexis at the other end of the communication.

2.3 CORBA

This section introduces the model elements of the CORBA metamodel and the artifacts produced by a CORBA application. Section 2.3.1 defines the CORBA object model. Section 2.3.2 introduces readers to the CORBA IDL, a language that used to specify the interfaces of CORBA objects. Section 2.3.3 illustrates the steps taken during a CORBA procedure call.

2.3.1 CORBA Object Model Concepts and Definitions

The object model we use is the model supported by the Portable Object Adapter (POA) and uses the static invocation mechanism. The POA model is a specialization of the general CORBA object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some additional components, and some of the components are defined more precisely than they are in the CORBA object model.

The abstract model supported by the POA has the following components [2] [20]:

- *Client*

A client is a computational context that makes requests on an object through one of its references.

- *Server*

A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.

- *Object*

In the model, *object* is used to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. From the server's view, the object identity is explicitly managed by object implementations through the POA interface. The object's interface is defined using the CORBA IDL, and the object's implementation is the servant.

- *Servant*

A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made regarding an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests regarding its references will be targeted at) multiple servants.

- *Object Id*

An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. It is specific to the POA

model. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

Note that the Object Id defined in the CORBA specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the `SYSTEM_ID` policy enables the POA to generate Object Id values for the application.

- *Object Reference*

An object reference is a handle used to identify, locate and address a CORBA object. To clients, object references are opaque entities. Clients use object references to direct requests to objects, but they cannot create or modify the contents of the object references. The POA model adds the constraint that a reference specifically encapsulate an Object Id and a POA identity.

- *POA (Portable Object Adapter)*

An object adapter is the link between the ORB and the object implementation. Its main responsibility is to manage the lifecycle of the CORBA Object, the Object Reference and the Object implementation. It is also responsible for mapping an object reference with the corresponding object implementation and method invocation. CORBA's POA has a standard interface definition and is an identifiable entity within the

context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.

- *Policy*

A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies that control the POA's threading model, as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.

- *The ORB*

As defined by the Object Management Group (OMG) in the *Object Management Architecture Guide*, the ORB provides the mechanisms with which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface [2]. An ORB's implementation is vendor specific, but the interfaces and functionality all conform to CORBA, a standard published by OMG.

- *CORBA IDL and IDL Compiler*

The OMG Interface Definition Language (IDL) is the language used to describe the interfaces that client objects call and object implementations provide. (See Section 2.3.2 for a simple sample IDL interface.)

An IDL Compiler takes IDL interfaces as input and produces artifacts as the bases for implementing the client and server side applications. The IDL Compiler is language specific as well as vendor specific, i.e., the generated artifacts may vary in number, content and programming language. However, they will all contain elements that are specified by the CORBA standard.

- *Skeleton*

The Skeleton is generated by the IDL compiler to be compiled and linked into the server application. It provides the run-time support required by the server application, allowing the operation invocations sent by clients to be received by the appropriate servant.

- *Stub*

A stub is a client-side function that allows a request invocation to be made via a normal local function call.

- *CORBA Data Type*

This refers to the language specific data type that are generated by IDL Compiler according to the language mapping rule.

- *Request*

A request is an invocation of an operation on a CORBA object by a client. Requests flow from a client to the target object in the server, and the target object sends results back in a response if the request requires one.

There are other components that belong to the POA object model but are not essential to the understanding of our work, such as the *POA Manager*, *POA Manager Factory* and *Servant Manager*. They are omitted here.

2.3.2 CORBA Interface Definition Language (IDL)

The contract between the server and the client of a CORBA object is the IDL interface of that object. IDL is programming language independent, and provides enough model elements for defining the interface of a CORBA object. There are standard mappings between CORBA IDL and all major programming languages such as C, C++ and Java. The implementation language of the server and client programs can differ as long as they use the same IDL interface of the object.

Following is a piece of the IDL interface from our case study—the Restaurant Food Ordering System (full details of this case study can be found in Section 5.1). The interface *KitchenTerminal* defines a CORBA object that provides some of the services that a restaurant kitchen could offer, such as allowing the client of the object (for example, customers or waiters) to query the current menu that this restaurant is offering. “currentMenu” and “newOrder” are two CORBA Operations. A CORBA operation can have parameters and return values as well as throw exceptions. There are three kinds of parameters: an *in*-parameter is only sent from client to server, while an *out*-parameter is only sent from server to client, and an *inout*-parameter goes both ways. The “newOrder”

operation of the `KitchenTerminal` has one in-parameter, specifying the contents of the order that is being sent to the kitchen.

```
interface KitchenTerminal{
    Menu      currentMenu (void);
    OrderID   newOrder(in Order anOrder);
    ...
};
```

The IDL also provides elements to define the data passed between the client and the server of an object. Besides the CORBA basic data types such as *char* or *float*, developers can also define data types that suit their application needs. The operation *currentMenu* has one return value *Menu*, which is a sequence of the user-defined CORBA Struct—*MenuItem*.

```
struct MenuItem{
    Dish   dishname;
    float  price;
};

typedefsequence<MenuItem> Menu;
```

Given the IDL, the programming language dependent IDL compiler can then generate the desired artifact, such as stub and skeleton, in the language of the server or client implementation.

2.3.3 CORBA Request Flow

In this sub-section, we use CORBA's C++ mapping to illustrate how a CORBA request is sent from the client to the server. Using static method invocation, the interface of a CORBA object is compiled by the IDL compiler into a stub and skeleton written in C++.

Figure 12 CORBA Request Flow

Once a client has obtained the object reference of the CORBA object, it could invoke CORBA operations on the CORBA object through the stub of that operation. For the client, the invocation is just like a local method call. The stub then forwards this request to the ORB of the client.

When the server's ORB receives such a request, it will delegate it to the POA that manages the servant implementing the CORBA object that the client wants to communicate with. The POA will then find the servant and dispatch the request to it through the skeleton.

The above is a simplified procedure of the CORBA's request processing using static method invocation and is the one we followed in our implementation in Chapter 5. This process may vary under different POA policies, and also differs when using dynamic method invocation.

2.4 Summary

In this chapter, we introduced the metamodels for both RoseRT and CORBA. RoseRT models are expressed by the UML-RT profile and CORBA models are expressed by the CORBA IDL. A couple of differences between the two metamodels are important for the work of this thesis. Hence it is worthwhile to emphasize them again.

- Differences in the request processing entity

In the CORBA model, CORBA objects are the entities that implement the application logic, while in the RoseRT model, capsules are the encapsulation of the application logic.

- Differences in method of communication

An application communicates with a CORBA object by invoking the operations defined in the IDL of the object. These method invocations are normally synchronous method calls. The RoseRT capsule communicates with its environments only by sending and receiving messages from its ports. Messages sent and received by RoseRT capsules are mostly asynchronous signals.

- Difference in message data

CORBA operations differentiate parameters according to their directions and are allowed to have multiple parameters. RoseRT signals are allowed to contain only one piece of data and no notion of out-parameters.

In light of these differences, we follow the MDA approach to accomplish the integration. For this purpose, we define a mapping that allows CORBA model elements to be represented by RoseRT models.

Chapter 3 RoseRT Profile for CORBA

3.1 Introduction

This chapter extracts a subset of the model elements defined in Rational RoseRT® and defines a CORBA profile that extends these model elements by adding stereotypes and tagged values. The goal of this profile is to establish an unambiguous bi-directional mapping between CORBA IDL and the structural model defined within RoseRT. This mapping will allow for the generation of CORBA IDL from RoseRT models and vice versa. Since CORBA IDL only defines interfaces, the structural model of the RoseRT application should only contain model elements that specify the interfaces of the system under development.

Figure 13 MDA Context of the RoseRT to CORBA Mapping

Figure 13 shows the MDA context of the mapping between RoseRT and CORBA. MDA relies on model mapping to transform a source model into a target model. The terms platform independent model (PIM) and platform specific model (PSM) are not very

clearly defined. In the context of this thesis, we deal with only one form of platform: distributed communication platform. We regard models that do not contain information specific to distributed communication technology as PIM. Therefore, the RoseRT model, in its native form, are considered platform independent in this thesis. CORBA as a distributed communication middleware is a PSM. Mapping between a CORBA model and a RoseRT model is a PIM to PSM mapping.

Figure 14 UML Profiles as Ways of Defining Model Mappings

As shown in Figure 14, to facilitate the mapping between a RoseRT model and a CORBA model, we define the *RoseRT Profile for CORBA*. This profile defines stereotypes that have specific meaning in a CORBA model. Using RoseRT profile to define a RoseRT model is adding constraints to the basic RoseRT model elements so that the RoseRT model contains enough information to be easily mapped into corresponding CORBA model.

In defining the profile, we try to achieve the following:

- Identify the subset of UML-RT entities that need to be extended in order to express a CORBA model.
- Identify the constraints required for the mapping to be consistent.

- Use examples to illustrate how the mapping should work.
- List the difference between the RoseRT Profile and the UML Profile.
- Discuss issues that arise when defining the mapping.

Our profile extends the following model elements in RoseRT: Class, Protocol, and Signal. This list is by no means exhaustive, but contains the minimal subset of concepts in RoseRT that are consequential to the mapping mentioned previously.

Each of the following subsections represents an entity defined in our profile. Within each subsection, the base RoseRT model element used to represent the CORBA type is introduced first. Then, the stereotypes, tagged values and constraints applied to the RoseRT model element to enable them to represent the CORBA types are explained. An example is then given to show the mapping between a piece of CORBA IDL code and the corresponding RoseRT class diagram. The structure of this section is very similar to that of the UML Profile for CORBA v1.0. Some examples of IDL code are obtained directly from the UML Profile for CORBA v1.0. We add a paragraph at the end of each subsection to describe UML and RoseRT differ in representing a given CORBA concept.

3.2 CORBA Struct

3.2.1 RoseRT Model Elements

3.2.1.1 Stereotype and Tagged Values

A CORBA struct is a user defined data type containing one or more named members of arbitrary type, including user-defined complex types. It is represented by a RoseRT class stereotyped <<CORBAStruct>>.

Each basic type member is represented as a RoseRT class attribute, and each user-defined-typed member is represented as an association.

Figure 15 Virtual Metamodel for CORBA Struct

3.2.1.2 Constraints

The representation of attributes of type CORBA Object defined in a CORBA struct is undefined in our model.

3.2.2 Notation

```
struct foo{
    long length;
    PropertyName name;
};
```

Figure 16 Struct Example

3.2.3 Differences from UML Profile for CORBA v1.0

None.

3.3 CORBA IDL Interface

3.3.1 RoseRT Model Elements

CORBA interfaces are modeled using RoseRT protocols.

- The representation for the IDL interface attribute is specified later in Section 3.3, “CORBA IDL Attributes”.

- The representation for IDL interface operations is specified in Section 3.5, “CORBA Operations”.
- The mapping from the parameter list of the CORBA operation to the data carried by the signals is defined in Section 3.4 “CORBA Struct Derived from CORBA Operations”

3.3.1.1 Stereotype and Tagged Values

An IDL interface is represented by a RoseRT protocol that is stereotyped <<CORBAInterface>>.

Figure 17 Virtual Metamodel for CORBA Interface

3.3.1.2 Constraints

- If in the IDL definition there is an attribute named “*attributeName*”, then there should not be an operation in the same IDL interface definition with either the name “*get_attributeName*” or “*set_attributeName*”, since both the attribute and the operation would be mapped to signals with the same direction and signal name, which is not legal in RoseRT.

3.3.2 Notation

```

interface TestInterface{

    struct TestStruct{
        string member1;
    };

    attribute TestStruct myStructAttr;

```

```

void myOp1 (in string str);
boolean myOp2 (in TestStruct t);
};3

```

Figure 18 Interface Example

3.3.3 Differences from UML Profile for CORBA v1.0

In the original UML Profile for CORBA v1.0, CORBA interface is modeled as a UML class stereotyped `<<CORBAInterface>>`. In the context of RoseRT, however, a class is normally used to express passive entity such as data storage. Capsule, on the other hand, is used to express an active entity that has its own thread of control and communicates with its environment by sending and receiving signals through its ports. The interface of the capsule, i.e. protocols of the capsule's ports, is more suitable to represent a CORBA interface than the interface of a RoseRT class.

In the UML Profile for CORBA v1.0, a CORBA struct defined within a CORBA interface is represented by an aggregation association between a `<<CORBAInterface>>` stereotyped class and a `<<CORBAStruct>>` stereotyped class. In our model, we represent the CORBA interface as a protocol. A protocol in RoseRT cannot be the aggregation end in an aggregation association. Therefore, a CORBA struct defined in a CORBA interface

³ Adopted from the sample IDL of UML Profile for CORBA v1.0 page 3-21.

is represented in RoseRT by a dependency relationship from a <<CORBAInterface>> stereotyped protocol to a <<CORBAStruct>> stereotyped class.

3.3 CORBA IDL Attributes

3.3.1 RoseRT Model Elements

Attributes defined in a CORBA IDL are represented using RoseRT signals defined in a RoseRT protocol. The reason for doing so is that CORBA IDL attributes are mapped to a pair of “get” and “set” operations during implementation. By CORBA standards, both operation names are identical to the original attribute name, but the get operation has a return value with the same type as the attribute and no parameter, and the set operation has one parameter with the same type as the IDL attribute and no return value.

Two RoseRT signals will be generated for the read operation of a CORBA attribute: one InSignal representing the request, and one OutSignal containing the result. For the write operation, only one InSignal will be generated, whose data will contain the new value of the attribute to be modified. For a read/write CORBA attribute, two InSignals and one OutSignal, defined in a RoseRT protocol, will be generated. In the RoseRT protocol definition, however, one cannot define two signals in the same direction with identical names, even if the two signals contain different data types. Therefore, the name of the InSignal corresponding to the read operation will be the original attribute name prefixed with “get_” and the name of the InSignal corresponding to the write operation will be the original name prefixed with “set_”. The name of the OutSignal,

since only the read operation will generate an OutSignal, will be identical to the original attribute name without any prefix.

A read only IDL attribute will be represented using a pair of an InSignal and OutSignal with exactly the same name as that of the original attribute.

3.3.1.1 Stereotype and Tagged Values

CORBA attributes are represented by RoseRT signals defined in a RoseRT protocol. The protocol must be stereotyped <<CORBAInterface>> and the signals stereotyped as <<CORBAAttribute>>.

Figure 19 Virtual Metamodel for CORBA Attribute

3.3.1.2 Constraints

One read/write attribute is represented by exactly three RoseRT signals, two InSignals and one OutSignal. The names of the two InSignals' must contain the common suffix of the original CORBA attribute's name as defined in the CORBA IDL.

One read-only attribute is represented by one InSignal and one OutSignal with their names identical to the original attribute name defined in CORBA IDL.

3.3.2 Notation

```
interface Vehicle{
    readonly attribute short tireCount;
    attribute PropertyName vehicleID;
};
```

Figure 20 Attribute Example⁴

3.3.3 Differences from UML Profile for CORBA v1.0

In UML Profile for CORBA v1.0, attribute representation differs according to the type of the attribute. Primitive types will be modeled as an attribute of a UML Class stereotyped as <<CORBAInterface>>. Attributes with a user-defined type are modeled by associations between the UML class and another UML class stereotyped as <<CORBAStruct>>. In our profile, the difference in attribute representation is caused not by the data type of the attribute, but by their read/write properties.

3.4 CORBA Struct Derived from CORBA Operations

3.4.1 RoseRT Model Elements

The representation of the parameters of an operation can be of several forms depending on the number of parameters and their kinds. Parameters in a CORBA operation can have three kinds: in, inout and out. Besides parameters, an operation can also have a non-void return value. There is no limit on either how many parameters an operation can have, or their kinds. A RoseRT signal, on the other hand, can contain only one piece of data, with no return value expected from any signals.

⁴ This diagram is not generated within RoseRT because RoseRT does not allow signals to be stereotyped.

To reconcile the differences between a CORBA operation's parameter list and RoseRT signal data, for each operation we construct two classes from the parameter list and return value according to the direction that the information flows. The data class that is passed with the InSignal contains attributes that are in or inout parameters from the CORBA operation, and its name is the operation's name suffixed with “_InStruct”. The data class passed with the OutSignal contains attributes that are inout and out parameters and return values for the CORBA operation; its name is the operation's name suffixed with “_OutStruct”.

If either data class contains only one attribute, then no class will be generated for that direction and the signal will contain data of the same type as the only attribute (or return value).

In any case, the name of the attribute that appeared in the constructed data class is either the same as the name of the formal parameter, or is named “ret_val” if the attribute represents the return value of the operation.

3.4.1.1 Stereotype and Tagged Values

Both data classes derived from CORBA operations are represented by RoseRT classes stereotyped <<OperationStruct>>.

Attributes in a <<OperationStruct>> stereotyped class must have one tagged value: kind. The tagged value can contain one of the following four strings: in, inout, out, return.

Figure 21 Virtual Metamodel for CORBA Struct Derived from CORBA Operation

3.4.1.2 Constraints

- <<OperationStruct>>-stereotyped classes must not have any operations.
- All attributes of a <<OperationStruct>>-stereotyped class must have visibility set to public.
- The name of the <<OperationStruct>> stereotyped class must be the same as that of the operation, suffixed by “_InStruct” or “_OutStruct” depending on the direction kind of the attributes contained in the class.

3.4.2 Notation

```

interface Banking {
    boolean login (in Card accessCard, in string password, out AccountInfo info);
    void withdraw (inout float amount);
};
  
```

Figure 22 CORBA Derived Struct Example

3.4.3 Differences from UML Profile for CORBA v1.0

Not in UML Profile for CORBA v1.0.

3.5 CORBA Operations

3.5.1 RoseRT Model Elements

Operations defined in the CORBA interface are represented by signals defined in RoseRT's protocols. In general, each operation will be represented by a pair of In- and OutSignals defined in the <<CORBAInterface>> stereotyped protocol. Both signals have the same name as the operation defined in the CORBA interface. If the operation is specified as oneway or otherwise contains no inout or out parameters or return values, in the corresponding RoseRT protocol definition the OutSignal will be omitted.

3.5.1.1 Stereotype and Tagged Values

CORBA's oneway operation will be represented as only one InSignal stereotyped <<CORBAOneway>> in the protocol definition.

Figure 23 Virtual Metamodel for CORBA Operation

3.5.1.2 Constraints

3.5.2 Notation

```
interface OpInterface{
    oneway onewayOp ( in string);
    boolean twowayOp ( in short);
};
```

Figure 24 Operation Example

3.5.3 Differences from UML Profile for CORBA v1.0

In the UML Profile for CORBA v1.0, an IDL operation is represented as a UML Operation of the <<CORBAObjectType>>-stereotyped UML Class that represents the containing CORBA interface. “The names and types of the operation parameters are mapped directly to the UML Operation Parameter names and types... [and] the IDL Parameter directional attributes “in”, “out” and “inout” are represented using the equivalent UML Parameter kind metaattribute.” [1]

3.6 Summary

This chapter defines the mapping between CORBA IDL and RoseRT model elements for key CORBA concepts, such as struct, interfaces, attributes, and operations. Since RoseRT only allows one piece of data to be passed along with any message, we also create a mapping between the formal parameters of a CORBA operation and a RoseRT data class. Since the UML representation of CORBA user defined data types is the same as RoseRT's representation, the mapping of CORBA user defined data types is omitted in our RoseRT Profile for CORBA.

Chapter 4 Requirement Analysis of the CORBA Proxy

Once we are equipped with a mapping between CORBA and RoseRT, we can design and implement a component that utilizes this mapping and helps integrate the RoseRT application with the CORBA platform. This component is called the CORBAProxy in this thesis. Before we going into the implementation details, we analyze the environment that the CORBAProxy will operate in and the requirements that the CORBAProxy must fulfill to accomplish its tasks.

Section 4.1 illustrates the context in which the CORBAProxy operates. Section 4.2 identifies the use cases that the CORBAProxy needs to support, and Section 4.3 explains the key concepts that the CORBAProxy must manipulate when enabling communication between a RoseRT application and a remote CORBA application.

4.1 The CORBAProxy in distributed RoseRT Communication

Figure 25 Architecture of Distributed RoseRT Application

Since we want the RoseRT model to be as independent of the CORBA platform as possible, we only allow the CORBAProxy to communicate with the AppCapsule directly using the RoseRT's signals; as such, the AppCapsule could operate in its native environment. The CORBAProxy also must communicate with the ORB, sending CORBA requests or receiving operation calls from the ORB. Figure 25 shows the request flow when a RoseRT application participates in the CORBA communication, either as a client or as a server or both. It differs from Figure 12 in two ways. First, the actual client or server of the CORBA communication is now a RoseRT application. Second, The RoseRT application communicates with the ORB through the CORBAProxy. Figure 25 only illustrates CORBA communication between two RoseRT applications, but also applies for cases when either side of the communication is not a RoseRT application. Since it is still a CORBA communication, we can substitute either the client or server in Figure 25 with the client or server from Figure 12, which includes a broad range of applications that CORBA could support.

Because the CORBAProxy is itself a RoseRT capsule, it is simple for it to communicate with the AppCapsule. However, it also needs to communicate with the ORB like any other ordinary CORBA application. When the AppCapsule needs to obtain services from a CORBA object, it sends signals containing the request to the CORBAProxy. The CORBAProxy will invoke corresponding CORBA operations on the object reference previously obtained during the registration phase of the CORBA object. The management of the object reference is CORBA specific; therefore, we assign such responsibility to the CORBAProxy rather than allowing the AppCapsule to handle it.

4.2 Use Cases for the CORBAProxy

In this section, we first identify the use cases and external actors that the CORBAProxy must satisfy in a use case diagram. Each external actor and use case is described in a separate subsection.

4.2.1 Use Case Diagram

Figure 26 Use Case Diagram

4.2.2 External Actors

There are four external actors that interact with the CORBAProxy: *RoseRTApp*, *ORB*, *RemoteCORBAApp*, and *RoseRTCORBAProfile*. The RoseRT Application

(*RoseRTApp*) uses CORBA as the DOC (Distributed Object Computing) middleware, and its communication with the CORBAProxy is mostly application related.

The *ORB* provides distributed communication services for the application, such as connection management, data transfer, parameter (de-)marshaling, endpoint and request demultiplexing, and so on. It shields the application from the details of the operating system, network platforms and hardware architecture environment. The CORBAProxy interacts with the ORB to obtain the connection and send and receive CORBA requests on behalf of the *RoseRTApp*.

The *RemoteCORBAApp* is the application that communicates with the *RoseRTApp* through the ORB. Although all CORBA requests are actually dispatched by the ORB, we distinguish between the two so that whenever the messages are application specific, we think of *RemoteCORBAApp* (instead of the ORB) as the actor of the use case because the former is actually the initiator or the final receiver of the CORBA requests. When the messages are services provided by CORBA to all applications, such as obtaining ORB references or sending requests to CORBA's Naming Services, we view the ORB as the actor.

The *RoseRTCORBAProfile*, which defines the mapping between the *RoseRT* and CORBA models, is also treated as an external actor in our design for the CORBAProxy. This actor separates the conversion of data representation between the two different object paradigms from the implementation of the CORBAProxy, so that the mapping itself can be as independent of the application as possible. This actor can be a subsystem, containing the definition of the mapping as well as the tool for transforming the CORBA

IDL into a RoseRT model and vice versa. The tool can be used by the CORBAProxy at runtime to obtain desired target data representation given the source. This separation of the model and data transformation rules from the entity that actually utilizes the transformation (the CORBAProxy in our model) allows the latter to be more independent of the technical details of the mapping, and thus more independent of changes to either meta-model and their mappings.

4.2.3 Use Case Descriptions

4.2.3.1 Establishing Connection with ORB

Use Case Identifier: ORBInitialization
Description: The CORBAProxy initializes the ORB runtime.
External Actors: ORB
Related Use Cases: None
Precondition:
Triggering event: Initialization of the CORBA Proxy.
<ol style="list-style-type: none"> 1. Initialize the ORB run-time environments. 2. Get references to ORB pseudo-object (for use in future ORB operations) and optionally other objects (including the root POA or some Object Adapter objects).[2]
Resulting event: The CORBAProxy obtains the reference to the ORB and POA.
Post condition: The CORBAProxy is ready for communication with distributed CORBA applications.
Alternatives: If the initial resolution of ORB reference is unsuccessful, then this process has to be repeated.
Nonfunctional requirements:
Comments: The CORBAProxy cannot provide any services before this use case is executed successfully.

4.2.3.2 Publish Services for the AppCapsule

Use Case Identifier: PublishService
Description: The CORBAProxy publishes services provided by the AppCapsule with the ORB, so that remote CORBA application could send CORBA requests to the AppCapsule
External Actors: ORB, RoseRTApp
Related Use Cases: None
Precondition: The CORBAProxy has obtained references to the ORB and POA.
Triggering event: A “register service” signal comes from the AppCapsule.
<ol style="list-style-type: none">1. Determine which IDL interface this service corresponds to.2. Find the servant implementation according to the IDL interface.3. Activate the servant.4. Publish this service, for example, register through the Naming Service.
Resulting event: Notify the AppCapsule that the service is published successfully.
Post condition: The AppCapsule is able to receive CORBA requests from CORBA clients.
Alternatives:
Nonfunctional requirements:
Comments: <ul style="list-style-type: none">• The mechanism used by the CORBAProxy to determine the corresponding IDL interface upon receiving the request from the AppCapsule is not specified by this use case.• The method used to publish the service is not specified by this use case.

4.2.3.3 Process Requests from CORBA Clients

Use Case Identifier: ProcessRequest
Description: When the AppCapsule functions as the server of a CORBA object, any request sent to it by a remote CORBA application will be processed by the CORBAProxy. The CORBAProxy is responsible for transforming CORBA requests into RoseRT signals, dispatching the signal to the correct RoseRT port and sending the reply back to the invoking client.
External Actors: ORB, RoseRTApp, RoseRTCORBAProfile, RemoteCORBAApp
Related Use Cases: None
Precondition: The AppCapsule is ready to receive requests; and a servant corresponding to this service is activated by the CORBAProxy.
Triggering event: A CORBA request is received by the CORBA servant.
<ol style="list-style-type: none">1. The CORBA request is transformed into a corresponding RoseRT signal.2. The RoseRT signal is sent to the AppCapsule through the appropriate port.3. Having handled the signal, the AppCapsule sends a signal containing reply information to the CORBAProxy.4. The information contained in the reply signal is used to construct the parameters and return values of the CORBA operation.5. The CORBA operation returns.
Resulting event: The invocation of the CORBA operation by the remote CORBA client returns.
Post condition: The request has been processed.
Alternatives: If the CORBA operation is a one-way operation, then step 3 and 4 above are omitted.
Nonfunctional requirements: None.
Comments: The mapping used to transform a CORBA request into a RoseRT signal and vice versa in step 1 and step 4 respectively is defined in Chapter 3.

4.2.3.4 Obtain Reference of Remote CORBA Object

Use Case Identifier: ObtainObjectReference
Description: Before RoseRTApp could send request to RemoteCORBAApp, the CORBAProxy has to obtain the object reference of the CORBA Object.
External Actors: ORB, RoseRTApp
Related Use Cases: None
Precondition: The CORBAProxy has obtained references to the ORB and POA.
Triggering event: The AppCapsule sends a request to the CORBAProxy, asking the latter to establish contact with the CORBA object to which it would send a request later.
<ol style="list-style-type: none">1. The CORBAProxy determines which CORBA object provides the service the RoseRT Application wants.2. The CORBAProxy obtains the object reference for such a CORBA Object.3. The correspondence between the CORBA object reference and the port of the AppCapsule, from which the 'establish contact' request is originally sent, is stored within the CORBAProxy.
Resulting event: A message is sent back to the AppCapsule, confirming that the contact with the remote CORBA Object is established.
Post-condition: Communication between AppCapsule and the remote CORBA object is enabled.
Alternatives: None.
Nonfunctional requirements: None.
Comments: None.

4.2.3.5 Send out CORBA Request to Remote CORBA Object

Use Case Identifier: SendOutRequest
Description: When RoseRTApp sends out a request to RemoteCORBAApp, the CORBAProxy needs to transform the RoseRT signals into a CORBA request and invoke the request on the object reference obtained previously. If any reply is expected, the CORBAProxy needs to construct signals from the reply and dispatch these to the correct port of the AppCapsule.
External Actors: ORB, RoseRTApp, RoseRTCORBAProfile, RemoteCORBAApp
Related Use Cases: None
Precondition: Communication between AppCapsule and the remote CORBA object is enabled.
Triggering event: The CORBAProxy receives a signal that represents a CORBA operation invoked on a remote CORBA object from the AppCapsule.
<ol style="list-style-type: none">1. Find out the reference of the remote CORBA object.2. Construct the CORBA request from the information contained in the RoseRT signal.3. Invoke the CORBA operation on the remote object.4. When the CORBA operation returns, construct the reply signal to the AppCapsule according to the information contained in the output parameter and return value of the CORBA operation.5. Send the signal to the AppCapsule.
Resulting event: The CORBAProxy sends the reply signal to the AppCapsule.
Postcondition: The CORBAProxy is ready to process the next request.
Alternatives: Step 4 and 5 may be omitted if the CORBA operation is defined as one-way in the CORBA IDL for the remote object.
Nonfunctional requirements: None.
Comments: The mapping used to transform a RoseRT signal into a CORBA request, and vice versa, in step 2 and step 4 respectively is defined in Chapter 3 RoseRT Profile for CORBA.

4.3 Concept Descriptions

Because the CORBAProxy is responsible for bridging two different object domains, it needs to interact with or manipulate entities that belong to both domains. This section highlights these entities' roles in participating in the CORBAProxy's work.

Figure 27 Concepts of the CORBA-distributed RoseRT Application

In Figure 27, we have two object domains. One is the traditional C++ CORBA model, and the other is the RoseRT application. In Chapter 2, we introduced concepts that are common to all CORBA applications, so in this section, we will introduce constructs that are specific to RoseRT applications.

One of the differences between a conventional RoseRT model and a RoseRT model designed to use CORBA is that the latter's interface is determined by the CORBA IDL that our RoseRT application has agreed on with the remote CORBA application.

Given the IDL interface, our RoseRT profile generates the following RoseRT model elements for the RoseRT application.

- **CORBA to RoseRT Mapping**

This mapping refers to the rules defined in the RoseRT Profile for CORBA.

- **Interface Protocols**

For each interface defined in the IDL, there is one <<CORBAInterface>>-stereotyped protocol generated for the RoseRT application capsule. The protocol's signals are defined according to the operations and attributes of the corresponding CORBA interface.

- **RoseRT Application Capsule (AppCapsule)**

The AppCapsule is the top-level capsule that represents the application that is designed and run within the RoseRT environment. In our context, this application communicates with other applications using CORBA as a distribution mechanism.

- **Service Ports**

To communicate with remote CORBA applications, our AppCapsule must have ports that are instances of interface protocols. These ports can be either a service-providing port, meaning that the AppCapsule is acting as the server for the CORBA object from whose interface definition the service port's protocol is generated, or it can be a service-requiring port, in which case the AppCapsule is the client of the CORBA object.

- **CORBAProxy**

The CORBAProxy encapsulate all the functionalities required for RoseRT applications to communicate with the ORB. It is designed as a RoseRT capsule. The name “proxy” comes from the Proxy Pattern [21], because this capsule represents either the remote target CORBA object or CORBA clients to the AppCapsule.

- **Proxy Ports**

To communicate with the AppCapsule, the CORBAProxy also has ports of the same <<CORBAInterface>>-stereotyped protocol with a conjugated protocol role. Message exchanges through such ports are CORBA operation related.

- **Data Types Generated from IDL Interface**

Data types can be generated from two kinds of sources in IDL interface. One kind is generated directly from CORBA user defined data types, according to the mapping defined by UML Profile for CORBA. The other is synthesized from the parameters and return values of the operations defined in an interface.

Chapter 5 Implementations of the CORBAProxy

In this chapter, we explain in detail the implementation of the CORBAProxy. Our goal is to provide an access to the underlying ORB that is as transparent as possible for the RoseRT implementation. We will use a case study to demonstrate the mapping and functionalities of the CORBAProxy and to illustrate and discuss the problems faced during implementation.

Section 5.1 starts by introducing the Restaurant Food Ordering System (RFOS) case study, giving the complete RFOS IDL interface definition. Section 5.2 gives the concept definitions of all entities used to implement the CORBAProxy, including the CORBAProxy's internal structure and all data structures used by it. Section 5.3 explains how the CORBAProxy realizes its responsibilities by showing the sequence diagram of each use case. The design decisions taken during the design and implementation of the CORBAProxy are discussed in detail. Section 5.4 discusses the configuration of the CORBAProxy for the RoseRT application capsule. The impact of the configuration on the application capsule is also discussed in this section.

5.1 Case Study: Restaurant Food Ordering System

5.1.1 Overview

Throughout this thesis, we use a simple restaurant food ordering system as a case study. We start by describing the functionalities provided by this system in Section 5.1.2, and then develop the interfaces of the system in IDL. The complete IDL specification of the RFOS system is given in Section 5.1.3.

5.1.2 Description of the Restaurant Food Ordering System (RFOS)

The RFOS allows the servers in a restaurant to order the food through wireless handheld devices (HHD) in the dining area. The ordered food then appears on a terminal in the kitchen (KT) so that the cooks know what they must prepare. Once the food has been prepared, the cook indicates the order is “ready for pick up” on the terminal inside the kitchen. The terminal then sends signals to the server’s handheld device so that he/she could deliver the food to the table.

To simplify our case study, we only consider the interfaces of the HHD that is used by the KT and vice versa. The interface of the HHD used by a server and the interface of the KT used by cooks or restaurant manager are omitted in our example. We could assume that those interfaces are local to the hardware devices and are implemented by the Java GUI.

5.1.2.1 Hand Held Devices (HHD)

A handheld device is carried by a server in the dining area. It provides a local interface to the server to edit order information for customers and a remote interface to the Kitchen Terminal to notify the server to pick up dishes that are ready to be served. Handheld devices come with a small amount of memory that holds a unique device ID. This ID is given during system configuration time and is assumed to remain the same during normal operation of the restaurant.

Handheld devices could receive a dish ready notification with the order id from the kitchen terminal when the order is ready. The server holding the device will then pick up the dishes and deliver them to the customers. The HandHeldDevice IDL is given below.

```

interface HandHeldDevice{
    void    pickUpOrder(in OrderID id);
};

```

5.1.2.2 The Kitchen Terminal (KT)

The kitchen terminal maintains information about all the orders in the restaurant and their status (for example, new, being prepared, ready and served etc.). It also contains the current menu of the restaurant, and provides the following operations for the remote handheld devices.

- **Current menu**

Upon request, the kitchen terminal sends out the current menu to a handheld device. When a handheld device wants to take a new order, it will first get the current menu from the kitchen terminal and then select items from the menu according to the instructions of the customers. Menu is an array of menu items, which in turn are the combination of the name and the price of a dish.

```

typedefstring Dish;
struct MenuItem{
    Dish    dishname;
    float   price;
};
typedefsequence<MenuItem> Menu;

```

- **Order**

After a customer places an order, which is an array of dishes, the server sends it to the kitchen using a handheld device. The kitchen terminal receives orders from handheld devices, displays them on its screen and returns a unique order ID to help the restaurant keep track of individual orders. Cooks select dishes to prepare from the list of pending orders shown on the kitchen terminal. When finished, cooks mark dishes as ready on

kitchen terminal. The latter will send a notification to the handheld device currently serving the table.

```
typedefshort OrderID;  
typedefsequence<Dish> Order;
```

- **Order status**

Sometimes customers want to know the status of their orders. For example, if the order hasn't been prepared, they might want to change some items in the order. The kitchen terminal can find out the status of a specific order. There are three states that are associated with any given order. An order is *NEW* when it has just been placed by the customer and sent to the kitchen terminal. It changes to *COOKING* when a cook picks this order from the new order list and starts preparing dishes associated with the order. It is *READY* when it is ready to be served to the customer.

```
enum OrderStatus {NEW, COOKING, READY};
```

5.1.3 The Complete IDL Specification

The following is the complete IDL for our RFOS.

```
module restaurant{  
    typedefshort OrderID;  
    typedefstring Dish;  
    typedefsequence<Dish> Order;  
    enum OrderStatus {NEW, COOKING, READY};  
    struct MenuItem{  
        Dish dishname;  
        float price;  
    };  
    typedefsequence<MenuItem> Menu;  
  
    interface HandHeldDevice{  
        void pickUpOrder(in OrderID id);  
    };  
  
    interface KitchenTerminal{  
        Menu currentMenu(void);  
    };  
};
```

```

        OrderID      newOrder(in Order anOrder);
        OrderStatus  orderStat(in OrderID id);
    };
};

```

Figure 28 and Figure 29 show the class diagrams of the data structure and protocols corresponding to the interface defined in IDL. The class diagram in Figure 28 follows the rules defined in the UML Profile for CORBA, and the protocol diagram in Figure 29 is derived by applying rules defined in the RoseRT Profile for CORBA.

Figure 28 RoseRT Data Types Correspond to CORBA Data Types Defined in the IDL

Figure 29 RoseRT Protocols Correspond to CORBA Interfaces Defined in IDL

5.2 Components of CORBAProxy

5.2.1 Servants

When another CORBA application need services from the CORBA object implemented by a RoseRT application, the ORB on the RoseRT server side can only dispatch such a request to a registered servant of that CORBA object. In an ordinary CORBA application, it is the servant that contains the implementation of the CORBA object's logic. In our situation, all the logic is implemented by the AppCapsule's state machine. Therefore, after the servant receives the upcall from the ORB, it must cooperate with the CORBAProxy for the latter to be able to send correct signals to the AppCapsule.

Figure 30 Servant as the Bridge between CORBA Operation and RoseRT Signal

Figure 30 modifies Figure 27 to show how the servant is used in our CORBAProxy implementation as the bridge between the CORBA object domain and the RoseRT object domain.

The servant is the last entity that the ORB and POA can reach while dispatching a CORBA request. It is also chosen as the starting point of the request processing within the RoseRT application. We define an abstract class *RTServant* to encapsulate the information required for the servant to determine the port/signal pair upon receiving a request from the ORB. Detailed information of *RTServant* class can be seen in Figure 34.

Our design is not the only way for to handle the dispatching of the CORBA request. A request reaches CORBA's object adapter (POA in our case) before it reaches the servant. If we are willing to modify the implementation of POA, we might be able to let it dispatch messages directly through RoseRT's ports. This approach achieves superior performance because there is one less node for a request to travel through on its path from the ORB to the AppCapsule. In doing so, however, we may become tightly coupled with the individual implementation of the ORB. Although the interface of the POA to the CORBA application is standardized, its interface to the ORB is not. The changes we make inside one ORB's POA may not be easily reusable for another ORB's POA. Since we do not presume which ORB our RoseRT application will work with, we decided not to modify anything within the ORB or POA, preferring to pay the performance penalty to achieve portability.

5.2.2 Capsules

5.2.2.1 CORBAProxy

The CORBAProxy encapsulates the ORB communication mechanism and allows the AppCapsule to communicate with remote CORBA applications in the same way it uses to communicate with another capsule. It runs on the same physical thread as the AppCapsule and communicates with the AppCapsule through wired ports.

Figure 31 Structure of the Restaurant System

In Figure 31, there are two bindings between the AppCapsule and the CORBAProxy. This is because the AppCapsule plays two roles in the restaurant system: it is the server of the CORBA object that implements the KitchenTerminal interface and it is also the client of the CORBA object that implements the HandHeldDevice interface.

The CORBAProxy contains an optional capsule role of type ORBContainer. In the initial transition of the CORBAProxy, it incarnates the ORBContainer on a different physical thread so that the latter can listen to the messages coming from the ORB.

Figure 32 Internal Structure of the CORBAProxy

5.2.2.2 ORBContainer

Figure 33 Structure of ORBContainer

The ORBContainer capsule is responsible for the registration and subscription of CORBA services, for constructing RoseRT signals upon receiving messages from an ORBWrapper, and for sending out CORBA requests to remote CORBA objects. The ORBContainer also keeps two lists of ports (or port instances) that are currently offering or subscribing to CORBA services.

5.2.2.3 ORBWrapper

The ORBContainer delegates the following responsibilities to the ORBWrapper: it initializes the ORB and allows the servant implementation to send RoseRT signals

converted from CORBA requests to the ORBContainer through the ORBWrapper's wired port to the ORBContainer.

In order for the AppCapsule to receive a CORBA request the same way it receives any other inter-capsule messages, we need to convert the CORBA request into RoseRT signals and send the latter to the AppCapsule through its public registered port. In our implementation, the servant, as the one receiving the ORB's up call, is responsible for the conversion. It will take the information in the CORBA request and construct corresponding RoseRT signals.

Ordinarily, RoseRT has an RTController for each physical thread that a capsule runs on. It is the RTController that handles the message passing between capsules. If the sending and the receiving capsules are run on the same thread, the same RTController is responsible for both the sending and the receiving. So, the receiver capsule will always know when someone sends it a signal. If the sender and the receiver are run on separate physical threads, there are two RTControllers; the sender RTController will put the signal in the receiver RTController's message queue and then signal the latter to pick up the signal.

Because the servant runs within its own thread and is not a RoseRT capsule, the signal constructed by it cannot be sent directly to the AppCapsule's port and be picked up properly by the AppCapsule's RTController. To overcome this problem, we allow the CORBAProxy to be on the same physical thread as the AppCapsule. However, it will contain an optional ORBContainer capsule that runs on a different physical thread and has an RTCustomController. The implementation of the RTCustomController enables a

CORBA servant to signal it just like any other RTController. The ORBWrapper has a port that connects to the ORBContainer. During the initialization of the servant, the reference of this ORBWrapper's port will be given and stored by the servant. Whenever a servant receives a request from the ORB, it will send a message through ORBWrapper's port to the ORBContainer and then signal the RTCustomController. The signaling is necessary because the servant runs within the ORB's thread that is different from the thread that the ORBContainer runs. Having been signaled, the RTCustomController will pick up the message and dispatch it to the ORBContainer. The ORBContainer will then construct a standard RoseRT message and send the request to the AppCapsule.

The reason we create a separate physical thread is to allow the AppCapsule to choose between using either the default RoseRT RTController or any RTCustomController it may want to use.

5.2.3 Protocols

There are three types of protocols needed by the CORBAProxy.

- FromORB

This protocol is used between ORBWrapper and ORBContainer and is application independent. The servant uses the port of the ORBWrapper to send messages to the ORBContainer when it receives a CORBA up call.

- ServerControl and ClientControl Protocols

These two application independent protocols contain the signals for the AppCapsule to register a port for a CORBA Object if the AppCapsule acts as the server

of that object, or to subscribe a port for a CORBA service if the AppCapsule is the client of the CORBA object.

- KitchenTerminalP and HandHeldDeviceP

These two protocols are application specific. They contain all signals that are derived from the two CORBA IDL interfaces KitchenTerminal and HandHeldDevice.

5.3 *Implementation of the Use Cases*

In this section, we use the RFOS case study to show the implementation of the use cases of the CORBAProxy.

5.3.1 Publish Service for the AppCapsule

When a RoseRT Application capsule provides certain CORBA services, it will implement all the related CORBA interfaces defined in a CORBA IDL. Its public ports then provide access points for all the CORBA operations defined in these interfaces. Its internal behavior should implement how each request is handled. In this section, we call CORBAPorts the ports associated with protocols of stereotype <<CORBAInterface>>. The CORBAPorts have the same functionality as other RoseRT ports except that they provide interaction points between the AppCapsule and remote CORBA Applications. From the RoseRT capsule's point of view, it receives requests and sends replies to its CORBA clients through these CORBAPorts just as it communicates with another RoseRT Application through ordinary RoseRT ports.

Figure 34 Class Diagram -- Publish Service

Figure 34 shows entities that related to the use case of PublishService. We've introduced the ORBContainer in subsection 5.2.2.2. The KitchenTerminalCap is the capsule that implements the logic of the control terminal inside the kitchen. It is the *AppCapsule* to the CORBAProxy.

The protocol KitchenTerminalP corresponds to the KitchenTerminal interface defined in the RFOS' CORBA IDL. All the signals exchanged between the AppCapsule and the CORBAProxy are application independent in this use case. They are specific for the registration of the service provided by CORBAPorts. We define a ServerCtrlProtocol to capture these interactions. We require all the service providing CORBAPorts to be subclass of this protocol, therefore the generalization relationship between KitchenTerminalP and ServerCtrlProtocol in Figure 34.

The CORBAPortDescriptor is a data structure used to store the information of a service providing port of the AppCapsule. An instance of the CORBAPortDescriptor class contains the name of the service that this port provides and a pointer to the port, among other things. The utility class ORBServices is used by the ORBContainer to establish communication with the ORB and manage CORBA requests during the runtime.

The KitchenTerminal_impl class is the servant implementation of the CORBA interface KitchenTerminal.

Figure 35 Sequence Diagram -- Publish Service

The AppCapsule will need to publish its services before any CORBA client can send requests to it. The unit of service publishing is each CORBAPort of the AppCapsule. The sequence diagram of this use case is shown in Figure 35. Whenever the AppCapsule (the KitchenTerminalCap in this case) is ready, it will send one “Register Service” signal through each CORBAPort to the CORBAProxy, together with a string of “Service Name”. This service name has to be agreed upon between the RoseRT Application and the application at the other end of the CORBA communication. The CORBAProxy will then forward this signal to the ORBContainer.

Having received this signal from the AppCapsule, the ORBContainer will instantiate the corresponding servant (object implementation) and register this servant through the Naming Service, using the service name contained in the signal data. If the registration is successful, the ORBContainer will store the information of this CORBA service, which includes the service name, the reference of its port from which it received the “Register Service” request and the reference to the instantiated servant, into a CORBAPortDescriptor. Having taken all the above steps, the ORBContainer will send a “Registered” signal to the AppCapsule (relayed by the CORBAProxy) and the latter then awaits requests from clients.

How does the ORBContainer know the class of the servant to instantiate when it receives a “Register Service” signal from the AppCapsule? In our implementation, the ORBContainer makes this decision according to the “Service Name” contained in the “Register Service” signal. The choice of the class of the servant is hard coded in the

transition trigger by the signal. For example, in the transition we may see the following code:

```
if (servicename = "Service1");  
    servant = new Class1();  
else if (servicename = "Service2");  
    servant = new Class2();
```

...

This hard coding, however, can be automated as long as we know the (service name, IDL interface name) pair before we generate the CORBAProxy (and the ORBContainer) from the CORBA IDL. One way to obtain such a pair implicitly is to impose the constraint that the "Service Name" used by the AppCapsule during registration has to be the same as the interface name defined in the IDL. Another more free way is to show the list of the IDL interfaces to the application developer before generating the CORBAProxy for the AppCapsule, and asking for the corresponding service name for each interface. By either method, we will have enough information to generate the source code for the CORBAProxy, like the example above.

- **Limitations of this implementation**

The AppCapsule has to use wired ports to connect with CORBAProxy. If there is more than one service provided by the AppCapsule, then more than one binding will exist between the AppCapsule and the CORBAProxy. The protocol of the AppCapsule must contain one more pair of signals than is needed by application itself, namely, registerService and registered. When RoseRT Applications communicate with each other using unwired ports through Connexis, there is a mechanism for each port to register its service using layer services. No explicit signal exchanges between the AppCapsule and other capsules are needed.

5.3.2 Process Request from CORBA Client

Before a CORBA client sends its request to a CORBA object, it will need to obtain the object reference of the latter. In our implementation, this process is done by resolving a service name through the Naming Service provided by the ORB. The same string for this service name is used by the AppCapsule during the registration of the service port. This string is stored in a CORBAPortDescriptor object in the ORBContainer together with the reference of the servant and the reference of the port that the ORBContainer used to communicate with the AppCapsule. The object reference obtained by the CORBA client is actually the reference of the servant. Unlike ordinary CORBA applications, this servant is not the one actually providing services. In our situation, the entity that provides services is the AppCapsule, through its public ports.

After the client has obtained the Object Reference, it can invoke operations on it. In an ordinary CORBA application, the call will be dispatched through the ORB to the servant, which will process the request and provide all the return information. Here we have two alternatives for implementation. The first is to leave the current ORB implementation intact and build an adapter for RoseRT application. The second is to modify one given ORB implementation, provide a POA that is specific for the RoseRT application, and hook the AppCapsule directly with the modified ORB. The adapter approach has the advantage of being able to communicate through any standard ORB implementation as long as it communicates with the ORB only through standard interface. The drawback is there is one more layer between the communicating CORBA client and CORBA server. The performance of this approach is therefore inferior to the alternative approach.

We choose the first approach in our implementation based on the above consideration, treating the ORB as a set of standardized interfaces. Therefore, we have to respect the operation dispatching mechanism defined in CORBA, allowing only the CORBA servants to receive CORBA requests from the ORB. The responsibility of the servant is to receive the up call from the ORB and dispatch this message to the AppCapsule.

Figure 36 Class Diagram – Process Request

Figure 36 shows the entities that are related to the use case of ProcessRequest. We've introduced the capsule ORBWrapper in subsection 5.2.2.3. The FromORB protocol defines signals exchanged between the ORBWrapper and the ORBContainer. The servant uses the ORBWrapper's "toProxy" port to send signals converted from CORBA requests to the ORBContainer. If there is a reply associated with the CORBA request, the servant will be added to a WaitingServantList as a blocked WaitingServant. The servant transforms the data received in a CORBA request and stores it in an object of CORBAData class. The ORBContainer will then construct signals according to the information stored in the CORBAData object and send them to the correct ports of the AppCapsule.

Figure 37 Sequence Diagram – Process Request

Figure 37 shows the sequence diagram of the ProcessRequest use case. Between the receipt of the CORBA request by the servant and the receipt of the matching RoseRT signal by the AppCapsule, a series of transformations occur. In the servant implementation for that CORBA operation, the servant will first transform the parameters of the CORBA operation into a RoseRT data type according to the rule defined in the

RoseRT Profile for CORBA. The servant will then generate a unique “Call ID” for this request and send a “CORBAMSG” signal through the “toProxy” port of the ORBWrapper along with a data object of type CORBAData. If the operation is of a synchronous type, the servant will add itself to a WaitingServantList with the Call ID and block itself until the reply comes back.

The port the servant used to send out the request is connected with the “fromOrb” port of the ORBContainer. The ORBContainer and the ORBWrapper run on their own physical thread and use a CustomController. Having received this “CORBAMSG” signal, the ORBContainer then finds the port by the service name and sends the signal through an idle port connecting to the AppCapsule. It then uses the Call ID as the flag of the port to indicate that this port is currently busy. The Call ID, port, and port index are then stored into an instance of CORBAPortDescriptor by the ORBContainer. After processing the request, the AppCapsule then sends the reply through the same port from which it received the request. The ORBContainer searches through its array of CORBAPortDescriptors to find the Call ID of the request to which this reply corresponds. With this Call ID, the ORBContainer first stores the data that came with the reply into a data structure and then wakes up the servant.

The servant, keeping its Call ID, is able to extract the reply data from the data structure where the ORBContainer last updated. It then transforms the data in the signal into parameters or return values of the CORBA operation and then returns. The CORBA client will then receive the return from the ORB.

We assume a CORBA synchronous operation for this use case because handling a synchronous call requires more effort from the CORBAProxy. If the operation is asynchronous, then no WaitingServant needs to be created and the use case finishes after signal 1.1.1.3 in Figure 37.

5.3.3 Obtain Reference of Remote CORBA Object

Before a CORBA Client can invoke operations on a remote CORBA Object, the former must obtain the object reference of the latter. To minimize the impact on the design of the AppCapsule, we assign the responsibility of resolving and maintaining the object reference to the CORBAProxy. This way, there is no additional data structure or attributes that need to be defined in the AppCapsule to store the reference.

Figure 38 shows the entities that participate in this use case. Data class BoundServiceList, BoundService and ServiceBoundPort are used by the CORBAProxy to store information of the object references of the remote CORBA objects and which ports (or port instances) on the AppCapsule are intending to invoke requests on these remote CORBA objects. The usage of these data classes is explained later when we discuss the sequence diagram of this use case.

Figure 38 Class Diagram -- Obtain Reference

One thing the AppCapsule needs to do, similar to what it does when using Connexis as the underlining communication mechanism, is to send a message to the CORBAProxy asking the latter to establish connection with the other party in the communication. Figure 39 below shows the request flow of this use case.

Figure 39 Sequence Diagram -- Obtain Object Reference

Along with the request, the AppCapsule will need to provide enough information for the CORBAProxy to determine what CORBA Object the AppCapsule would like to communicate with. In our implementation, the object reference is obtained by calling the ORB's Naming Service. So, we require the AppCapsule to provide in its "bind service" request a literal string, which represents the name of the service provided by the CORBA Object.

Figure 40 BoundServiceList

After the Naming Service resolves the object reference from the service name, the CORBAProxy will create a new BoundService object, as shown in Figure 40, which stores the following information:

- The port index of the CORBAProxy port from which it receives the AppCapsule's request;
- The Object Reference of the remote CORBA Object;
- The name of the service.

The CORBAProxy maintains a list of such BoundService objects, called the BoundServiceList, using the port through which the request is received as the index into these objects.

- **Implications of our implementation**

One port instance can be associated with only one object reference. Since the data structure BoundServiceList used by ORBContainer searches the object reference by the port instance, our implementation does not allow the AppCapsule to send requests to different remote objects through the same port. A closer examination of a RoseRT port that is used to send out CORBA requests reveals that the port instance's role is like that of the CORBA stub used by a CORBA client. Both provide a location transparent handle for the client to access the remote object. If the AppCapsule wants to communicate with more than one CORBA object during its lifetime, it will have to use separate ports for each object. This is analogous to an ordinary CORBA client process obtaining a distinct stub for each object it wants to invoke requests on.

After the ORBContainer returns the “service bound” signal, all signals coming from this port instance of the AppCapsule will be interpreted as CORBA requests to the particular object.

5.3.4 Send out CORBA Request to Remote CORBA Object

Figure 41 Class Diagram -- Send CORBA Request

After the AppCapsule has obtained the reference of the remote object, it can invoke operations on it. As described in the implementation of the “obtain reference” use case, the AppCapsule associates one of its ports with a particular CORBA object. When the AppCapsule wants to invoke a request on this object, it will send a RoseRT signal through that port.

Figure 42 Sequence Diagram -- Send CORBA Request

Having received this signal, the ORBContainer first goes through its BoundServiceList to find the BoundService for the port instance from which the signal came. The reference of the remote object is then extracted from the BoundService data structure. The ORBContainer then constructs and invokes the CORBA request. When control returns from the ORB, the ORBContainer constructs a reply signal to the AppCapsule.

In the above explanation, we use an example of a CORBA synchronous operation call: pickUpOrder to illustrate how the CORBAProxy works. If the request sent out by the RoseRTAppCapsule is an asynchronous one, the this use case could end after signal 1.3 in the above figure.

The rules used by the ORBContainer to construct a CORBA request given a RoseRT signal and construct a RoseRT signal according to the return of the CORBA operation is defined in Chapter 3.

Implications of our implementation

The implementation of the ORBContainer requires the sequential processing of client requests. This is because during the transition triggered by a signal containing a CORBA request, once the ORBContainer invokes the request through the stub, the control is obtained by the ORB and will only be returned after the remote object finishes processing the request and the CORBA operation returns. During this process, the ORBContainer is blocked waiting for the reply. If the AppCapsule invokes another request, either on the same object or a different one, the signal will be queued and processed after the previous request returns.

This implementation certainly limits the concurrency of the client process. There are several ways to improve the situation. One is to implement a thread pool for the ORBContainer, allocating one thread for each client request. Note that if the client tries to invoke multiple requests on the same object, the behavior is determined by the policy used at the end of the server process.

Another way to improve the concurrency of the ORBContainer is to use the CORBA dynamic invocation.

Since we have a mapping from the CORBA interface to the RoseRT protocol, the capsule that contains the port of the protocol will be able to handle all the requests. However, currently we have no mechanism to ensure all CORBA operation are

implemented by the AppCapsule; in other words, there is a transition associated with each CORBA operation defined in the state machine of the AppCapsule. In ordinary CORBA applications, the object implementation must provide definition for all CORBA operations defined in the IDL. An incomplete object implementation cannot be instantiated by the server process. In our environment, we do not have such a constraint when instantiating the AppCapsule. Theoretically, before we instantiate the AppCapsule, we could check all the ports whose protocol is stereotyped as CORBAInterface and see if any signal stereotyped as CORBAOperation appears as a trigger event for some transition defined in the state machine of the AppCapsule.

Suppose the two communicating parties of a CORBA application are both RoseRT Applications and are of peer-to-peer relationship, i.e., both require services from the other party. This relationship is not clearly shown by the IDL interface, which will only contain two unrelated interfaces, each related to services provided by a single RoseRT Application. If during the development of the implementation, we define the CORBA IDL first, with two interfaces, then later when we design the RoseRT application, we will generate two separate protocols. Each RoseRT AppCapsule will then contain two ports, one to implement the services that the AppCapsule provides and call “register service” to publish services, and the other to contain the services that the AppCapsule requires from the other party and will “bound service” to obtain the object reference. Conceptually, two bindings exist between the two AppCapsules.

If, during the development of the CORBA application, we design the two AppCapsules first, it is natural that there will be only one binding, and hence one

protocol, existing between them. Each AppCapsule will have only one port. Signals going out from the port are requests from this AppCapsule to the other. Signals coming in from the port represent services provided by this AppCapsule. We still need to generate IDL interfaces for this application. One easy way is to put all in-signals of the protocol into one interface and all out-signals into another. By the asynchronous nature of the RoseRT signals, all the operations defined in the two interfaces will be one-way operations.

Although in this case we still have two interfaces defined in CORBA IDL, we do not want to create two different protocols again and add an extra binding between the two AppCapsules. Since one goal of our implementation is to minimize AppCapsule modification during configuration for CORBA communication, we would like to keep the original protocol and binding between the two AppCapsules. The port on each AppCapsule now plays two roles in the CORBA communication: it both receives and sends out CORBA requests. Therefore, the same port will send both “register service” and “bind service” requests to the CORBAProxy. One problem arises from this arrangement. When the CORBAProxy receives a signal from the AppCapsule later, how can it determine whether the signal is a request to a remote CORBA object or a signal corresponding to the reply of a previous CORBA request invoked by a remote client? This problem has to be handled by the CORBAProxy. Each time the CORBAProxy receives a signal other than “register service” and “bind service” from the AppCapsule, it will first search through its list of pending replies. A pending reply will exist if a signal that corresponds to a synchronous CORBA operation has been sent through this port

previously and no signals corresponding to the reply of that operation have been received yet. If this signal contains the reply to some CORBA operation, the CORBAProxy will wake up the servant and the latter will finish the CORBA operation. If the signal matches on reply, the signal will be processed as a CORBA request.

In the case where the IDL interfaces are generated from the RoseRT protocol, since all the operations are one-way, there will never be any pending reply in the CORBAProxy. Therefore, any signals sent out by the AppCapsule will be treated as CORBA client requests, which is exactly how they should be treated.

5.4 Configure a RoseRT Application for Distribution

A RoseRT Application may reach the point of being ready to configure for distribution from two different paths. One path starts with a CORBA IDL interface and no RoseRT capsules. The other starts with a more or less stable RoseRT Application (by stable, we mean that the protocols used by communicating capsules and data structures passed along with signals are determined) and no IDL.

If we came along the first path, the public interfaces and possibly some data structures of the RoseRT Application will already be defined even before the AppCapsule has come into existence. The IDL Compiler will generate protocols and data structures (either directly from IDL definitions or constructed from the parameters of CORBA Operations in a CORBA interface) from the CORBA IDL, according to the rule specified in the RoseRT Profile for CORBA. The task of the designer of the application is to create the internal structure and behavior of the component to implement the services the

AppCapsule should provide, or utilize the services provided by the environment, and set the visibility of the port that either receives or sends CORBA requests as public.

If we came along the second path, the RoseRT Application should still have the protocols of all its public ports and related data structures ready. This time, however, we need to generate CORBA IDL before our AppCapsule can communicate with remote CORBA applications.

To sum up, two conditions must be met before we can proceed to the configuration phase: we must have both the IDL for the CORBA application and all the CORBA ports of the AppCapsule ready. When both conditions are satisfied, we can create a new top-level capsule and put our AppCapsule inside it. The next step, then, is to generate a CORBAProxy for our RoseRT Application.

5.4.1 Generate the CORBAProxy

In our implementation, we have a different CORBAProxy for each different application. The differences in the CORBAProxy appear in two places: the number and protocol of ports used to communicate with the AppCapsule, and the generated CORBA Object implementations. If, for example, the AppCapsule provides services defined by two separate CORBA Interfaces and therefore has two service-providing CORBA ports, the CORBAProxy will have to contain two ports with compatible protocols, each connecting with one of the AppCapsule's CORBA port.⁵ In the same situation, our

⁵ The improvement of the CORBAProxy to make it independent of application is discussed in the future work section.

implementation will have to generate two separate object implementation classes, one for each IDL interface implemented by the AppCapsule.⁶

When generating the CORBAProxy, we need to follow several steps:

1. **Specify the IDL for the whole application and assign a service name to each interface defined in the IDL.**

The interface and service name pairs are stored in the CORBAProxy. Later, when the AppCapsule sends a “Register Service” signal to the CORBAProxy, the latter will extract the service name contained in this signal and know right away the name of the IDL interface. The CORBAProxy can then instantiate the corresponding CORBA Object implementation according to the relationship between the name of the interface and the name of the class of the object implementation.

2. **Generate the ports of the CORBAProxy that connect to the CORBAPorts of the AppCapsule.**

As the precondition of the configuration phase, we require the existence of the public CORBAPorts of the AppCapsule. Therefore, when we want to generate a CORBAProxy with a matching structure, we could create ports of the CORBAProxy based on the CORBAPorts of the AppCapsule. We only need to use the protocol role that is the conjugate of the role of the CORBAPort.

The existence of this step marks a deficiency of our implementation compared with Connexis, whose capsule responsible for remote communication is independent of individual application (i.e., the number and protocol of the unwired ports it supports). To do this, we must be able to create a port on the CORBAProxy that can send and receive

⁶ The possibility of generating only one generic object implementation for all CORBA applications is discussed in the future work section.

any signal from any port. It would certainly be an improvement to our implementation if the structure of the CORBAProxy were independent of individual applications.

3. Generate the CORBA Object's implementation classes.

We need to generate one class for each object implementation of an IDL interface. The name of the class will be the name of the interface plus the suffix “_impl”. Therefore, once we know which CORBA interface our AppCapsule will implement, we will know the name we should give to the class of objects implementing the interface.

According to our mapping from CORBA IDL to RoseRT protocols, we know that the name of the protocol is the same as the name of the IDL interface. We also know that a port with the base role of a CORBAInterface stereotyped protocol must be a service providing port, and a port with a conjugate role of such a protocol must be a service requiring port. Therefore, after we use the protocol role information to generate the ports of the CORBAProxy, we can use the same information to decide which IDL interfaces our AppCapsule is providing and which are required. Given the name of the protocol, and hence the name of the IDL interface, we can deduce the name of the skeleton class from which this implementation class is to be derived and the name of the implementation class itself. The information of the methods this class should implement is contained in the IDL definition of the interface.

Note that at this time, there is no explicit relationship between the CORBA object's implementation and the port on the CORBAProxy through which the implementation will dispatch the CORBA operation's up-call received from the ORB. Although there is a relationship between the two, since both the name of the protocol of

the port and the name of the object implementation can be deduced from the name of the IDL interface, we cannot rely solely on this implicit relationship. C++, for example, does not have reflection for us to instantiate and we will need to use the Reflection Pattern to provide such a mechanism.

5.4.2 Structural changes

In our implementation, when configuring the AppCapsule for distribution, we need to perform the following steps to modify the structure of the AppCapsule:

1. Create a top-level capsule, and put both the AppCapsule and the CORBAProxy inside it.
2. Connect the AppCapsule's CORBAPort with the corresponding ports of the CORBAProxy.

- **Comparison with Rational Connexis**

When using Connexis as the middleware for distributed RoseRT Application communication, we need to first configure the capsule for Connexis. This step adds a capsule inside the AppCapsule, with the former providing all the mechanisms necessary for the latter to engage in remote communication with other RoseRT Applications. The AppCapsule in this case remains the top level capsule of the component as a unit of distribution.

The port used by the AppCapsule for remote communication becomes an unwired port. One of the two communicating ports will need to “publish” its service. This is similar to the “register service” signal in our implementation.

5.4.3 Behavioral changes

The AppCapsule must modify its state machine to add the behavior of sending “register service” or “bind service” signals to the CORBAProxy before accepting or sending CORBA requests. This is like calling “registerSPP” or “registerSAP” operations on unwired ports when, using Connexis, the designer chooses to register services for ports by application.

5.5 Testing of the CORBAProxy

The case study introduced in this chapter is also used to test the correctness of both our RoseRT profile and the behavior of the CORBAProxy. We use the ACE ORB (TAO) as the ORB of RoseRT application.

For the testing of use cases and the mapping defined in RoseRT profile for CORBA, two C++ RoseRT applications are developed: one for the KitchenTerminal and the other for HandHeldDevices. All operations defined in the IDL of the RFOS are tested to insure the correct operation of the CORBAProxy.

For the testing of inter-ORB communication, a Java application implementing HandHeldDevices is implanted and the Orbacus’ Java ORB is used.

Further testing need to be done against more diversified applications.

Chapter 6 Conclusion and Future Work

6.1 Conclusion

The goal of this thesis is to enable RoseRT applications to use CORBA as the distributed communication middleware. To do so, we first defined a mapping between model elements of CORBA IDL and RoseRT. The CORBAProxy capsule is then

designed and implemented to allow RoseRT applications to communicate with remote CORBA applications, using the mapping as the rules of transforming CORBA operation calls into RoseRT signals and vice versa.

Our implementation of the CORBAProxy also satisfies the following conditions:

- The time of executing the integration process is flexible. It could be before, during or after the development of the AppCapsule.

If before the design of the AppCapsule, the development team decides to use CORBA as the communication platform, they can design the CORBA IDL first. The corresponding RoseRT protocols and data classes can be generated from the IDL. Then, the RoseRT developer can implement the AppCapsule using the generated protocols and data classes.

If, after the design of the AppCapsule, the development team decides to integrate with CORBA, they could start by specifying which port is going to participate in the CORBA communication. CORBA IDL will be generated according to the protocol of ports with the appropriate user defined CORBA data struct.

To sum up, as long as we have both the CORBA IDL and the corresponding RoseRT protocol data type definition, we can complete the integration process by adding the CORBAProxy into the RoseRT application.

- The RoseRT application should be able to communicate with distributed applications in the same way it would communicate with another RoseRT application.

Since the CORBAProxy is designed within RoseRT and communicates with the AppCapsule only by sending and receiving messages through its ports, the AppCapsule will communicate with remote CORBA applications in exactly the same way as with another RoseRT application.

- The integration process should have minimal impact on the structure and behavior of the RoseRT application.

During the integration process, we need to add the CORBAProxy capsule into the RoseRT application and we need to modify the state machine to allow the registration of the ports of the AppCapsule that participate in the CORBA communication. This process is similar to what currently exists with Rational Connexis, and is necessary for an application to prepare for distribution. There is no other impact on the AppCapsule's structure or behavior.

6.2 Future Work

- **Implement the generator for the CORBAProxy**

In our implementation, we follow the mapping defined in Chapter 3 to generate RoseRT protocols and data classes from CORBA IDL interface. The number of ports that the CORBAProxy contains is in turn determined by the total number of CORBA interfaces that the AppCapsule provides and requires. The protocol role of each individual CORBAPort is determined according to whether the AppCapsule is providing a CORBA interface or receiving it.

All the above information is implementation specific, differing according to the definition of the IDL interface. In our implementation of the CORBAProxy, it is handled

by inspection. RoseRT models are stored in textual format. Each model element, such as the structure of a capsule, has a well-defined representation format. It would be possible then to implement a generator for the CORBAProxy that would automate the process, i.e. that would read in the information defined in the IDL interface and add the desired ports to the generic structure of the CORBAProxy.

- **Create a more generic CORBAProxy**

The structure of the CORBAProxy is currently implementation dependent in that the number of CORBAPorts and their role is determined by the IDL interfaces that the AppCapsule requires or provides. We could create a generic port for handling all the CORBA related messages for the AppCapsule.

- **Use a default servant to implement all the CORBA objects**

In our implementation, we generate one servant implementation class for each CORBA object for which the AppCapsule acts like the server. We could implement one default servant and by using the USEDEFAULTSERVANT policy of the POA, we could register this servant to receive all the ORB upcalls.

- **Check the consistency of the CORBA to RoseRT mapping**

In this thesis, we only designed a mapping between the RoseRT metamodels and the CORBA IDL interface. Further work needs to be done to check the consistency of this mapping.

- **Test the performance of the CORBAProxy**

The performance testing of the implementation can have at least two approaches. Firstly, when both parties of the communication are RoseRT applications, we need to

compare the performance our ORB-based implementation against Rational's Connexis. Secondly, we need to do simulated testing on the CORBAProxy to see how it performs.

- **Generalize the work of this thesis for UML 2.0 and CORBA 3.0**

We believe that the result of this thesis could provide an important contribution towards the definition of UML 2.0 Profile for CORBA 3.0. For example, the concept of ports appeared in UML2.0 was not in UML1.x, but was already included in the RoseRT model.

- **Toward industrial solution**

There are quite a few issues related with integrating CORBA with RoseRT. This thesis proposed solutions to some of the main issues. The implementation in this thesis is more a proof of concepts than an attempt to build an industrial solution. In order for the CORBAProxy to meet industrial requirement, much work still needs to be done in the future. For example, we would like the CORBAProxy to be able to support CORBA facilities such as the event service and trading service. We would also like the CORBAProxy has the ability to behave accordingly to the different policies that the ORB is using during the run-time.

Bibliography

- [1] Object Management Group, *UML Profile for CORBA v1.0*,
http://www.omg.org/technology/documents/formal/profile_corba.htm.
- [2] Object Management Group, *Common Object Request Broker Architecture (CORBA/IIOP)*, v3.0.2,
http://www.omg.org/technology/documents/formal/corba_iiop.htm.
- [3] Object Management Group, *CORBA Component Model v3.0*,
<http://www.omg.org/cgi-bin/doc?formal/02-06-65.pdf>.
- [4] Object Management Group, *Common Warehouse Metamodel™ (CWM™) Specification*, v1.1, <http://www.omg.org/technology/documents/formal/cwm.htm>.
- [5] Object Management Group, *Meta-Object Facility ((MOF™))*, version 1.4,
<http://www.omg.org/technology/documents/formal/mof.htm>.
- [6] Object Management Group, *UML 2 Infrastructure submission*,
<http://www.omg.org/cgi-bin/doc?ad/2003-03-01.pdf>.
- [7] Object Management Group, *UML 2 Superstructure Final Adopted Specification*, <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>.
- [8] Object Management Group Architecture Board, *Model Driven Architecture*,
OMG whitepaper, <http://www.omg.org/docs/ormsc/01-07-01.pdf>.
- [9] Douglas C. Schmidt, Frank Buschmann, *Patterns, frameworks, and middleware: their synergistic relationships*, Proceedings of the 25th international conference on Software engineering. p. 694 – 704, 2003.

- [10] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Nanbor Wang
Applying Model-Integrated Computing to Component Middleware and Enterprise Applications, The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules, vol. 45, p. 65 – 70, Oct. 2002.
- [11] Jan Hendrik Hausmann, Stuart Kent, *Visualizing model mappings in UML*, Proceedings of the 2003 ACM symposium on Software visualization, p. 169 – 178, 2003.
- [12] David Akehurst, Stuart Kent, *A relational approach to defining transformations in a metamodels*, In UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany. Proceedings, Springer, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds., vol. 2460 of LNCS, p. 243-258, September/October 2002
- [13] Gregor Engels, Reiko Heckel, Jochen M. Kuster, and Luuk Groenewegen, *Consistency-preserving model evolution through transformations*, In UML 2002 – The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany. Proceedings, Springer, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds., vol. 2460 of LNCS, p. 212-226, September/October 2002.
- [14] Johanna Ambrosio, *Tools for the code generation*,
<http://www.adtmag.com/article.asp?id=7850>.
- [15] Darrell Brunsch, Carlos O’Ryan, Douglas C. Schmidt, *Designing an Efficient and Scalable Server-side Asynchrony Model for CORBA*, Language, Compiler and

- Tool Support for Embedded Systems: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, p. 223 – 229, 2001.
- [16] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran, *Evaluating Meta-Programming Mechanisms for ORB Middleware*, IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies, p. 102 – 113, October 2001.
- [17] Wolfgang Emmerich, *Distributed component technologies and their software engineering implications*, International Conference on Software Engineering Proceedings of the 24th international conference on Software engineering Orlando, Florida, P. 537 – 546, 2002.
- [18] Rational Software, *Rational RoseRT C++ Reference*, 2002.
- [19] Anneke Kleppe, Jos Warmer, Wim Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*, Addison Wesley, 2003.
- [20] Michi Henning, Steve Vinoski, *Advanced CORBA Programming in C++*, Addison Wesley, 1999.
- [21] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.