A Case Study on the Model Driven Architecture: Control and Management of Audio and Video Streams

Thesis for a Master of Science degree in Computer Science from the University of Twente

David Baakman

Enschede, October 2003

Graduation Committee:
Prof. dr. ir. L.J.M. Nieuwenhuis
J.P. Andrade Almeida, M.Sc.
Dr. L. Ferreira Pires
Dr. ir. M.J. van Sinderen
Acknowledgements

Working on a project such as this for several months the outcome could not be possible without the help of many other people.

I would like to thank my supervisors, João Paulo Andrade Almeida, Luís Ferreira Pires, Marten van Sinderen and Bart Nieuwenhuis. Their numerous comments on this report and my other work helped me a lot.

Furthermore, I would like to thank the people at the Architecture group of the Computer Science department of the UT. The discussions with them, whether they were held during lunch, a tea/coffee break, or just during the day, were always interesting.

My sister, Tamara, helped me with the final proofreading, thanks for that. Finally, I want to express my deepest gratitude to my parents. They have made quite some offers in the past to ensure that I was able to study and become the person who I am today. I am very grateful for that.
Abstract

The Model Driven Architecture (MDA) has been introduced by the Object Management Group (OMG) in order to be able to model systems independently of the (middleware) platform used. For this, Platform Independent Models (PIMs) and Platform Specific Models (PSMs) are defined in MDA. PIMs model a system independently of a set of possible target platforms and PSMs model a system in terms of a particular platform.

When the transformation between PIMs and PSMs are well understood, the development of distributed applications can become less costly.

This thesis aims at defining what information should be present in PIMs and PSMs, particularly considering the use of UML as modeling language. This is done by following an MDA trajectory for a case study. This case study includes a PSM to PIM transformation and a PIM to PSM transformation, exercising different kinds of relations between platform-specific and platform-independent models. Furthermore, the case study involves PSMs for two different platforms, namely CORBA and Web Services.

The case study consists of porting an application that makes use of the OMG Audio/Video Stream Specification on the CORBA platform to an application on the Web Services middleware platform. For this, we have first modeled the existing application resulting in a CORBA PSM. After that, this CORBA PSM has been transformed into a PIM. Finally, a Web Services PSM has been made.

In the PIM, we have used the Object Action Language (OAL) for specifying actions in UML statechart diagrams. Furthermore, an abstract component diagram is introduced in order to group various interfaces together in order to create coarse-grained services.

We can conclude that it is possible to use MDA for porting an application from one middleware platform to another. However, the overhead introduced by defining the various intermediate models and by having to define a mapping between these models makes this approach beneficial only when multiple applications have to be ported from one platform to another.
# Table of Contents

1. **Introduction**  
   1.1 Background ................................................................. 7  
   1.2 Goals ........................................................................... 8  
   1.3 Approach ................................................................. 8  
   1.4 Structure ................................................................. 9  

2. **Model Driven Architecture**  
   2.1 Overview ................................................................. 10  
   2.2 MDA Core and Middleware Technologies .................. 11  
   2.2.1 UML ................................................................. 11  
   2.2.2 MOF ................................................................. 11  
   2.2.3 CWM ................................................................. 12  
   2.2.4 Middleware Platforms ........................................... 12  
   2.3 Models ................................................................. 12  
   2.3.1 Platform Independent Model .................................... 14  
   2.3.2 Platform Specific Model .......................................... 15  
   2.4 Transformations ...................................................... 15  
   2.5 MDA Projects ......................................................... 16  

3. **Case Study: Audio/Video Stream**  
   3.1 OMG Audio/Video Stream Specification .................... 18  
   3.1.1 Motivation .......................................................... 18  
   3.1.2 The Objects ........................................................ 19  
   3.2 Initial Implementation ............................................... 21  
   3.3 The Trajectory .......................................................... 22  
   3.3.1 The Phases ........................................................ 22  
   3.3.2 The Scope .......................................................... 23  

4. **CORBA Specific Models**  
   4.1 Overview ................................................................. 24  
   4.2 Interface PSM .......................................................... 25  
   4.3 CORBA Objects PSM Static View ............................... 29  
   4.4 CORBA Objects PSM Dynamic View ......................... 31  
   4.4.1 Sequence Diagrams ............................................... 32  
   4.4.2 Statechart Diagrams .............................................. 35  
   4.4.3 Discussion .......................................................... 37  
   4.5 Realization PSM ....................................................... 38  

5. **Platform Independent Model**  
   5.1 Static View ............................................................. 41  
   5.1.1 The Classes ....................................................... 41  
   5.1.2 Properties .......................................................... 42  
   5.1.3 Abstract Components ......................................... 44  
   5.2 Dynamic View ....................................................... 46  
   5.2.1 Object Action Language ....................................... 46  
   5.2.2 Statechart Diagrams ........................................... 47
6 Web Services Specific Models

6.1 Interface PSM

6.1.1 WSDL and a WSDL UML Profile

6.1.2 Mapping RemoteObjects, operations and attributes

6.1.3 Mapping Identifiers

6.1.4 Mapping the State

6.1.5 Resulting Class Diagram

6.1.6 Dynamic View

6.2 Realization PSM

7 Evaluation

7.1 Differences between PIMs and PSMs

7.2 The usage of UML

7.3 The usage of OAL

7.4 The usage of Components

7.5 The usage of EDOC

7.6 The usage of Transformations

8 Conclusions

8.1 Contributions

8.2 General Conclusions

8.3 Future Work

Abbreviations

References

Appendix A: Additional Diagrams

CORBA PSM Diagrams

PIM Diagrams

Web Services Diagrams
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>The trajectory</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Dependencies between the CIM, PIM and PSM in a top-down methodology</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Refinement of PIMs</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>PSM becoming a PIM</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Model Transformation</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Types of interaction</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Examples of streams</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>The Stream Endpoints and their connections</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Objects used in the specification</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Construction of the system</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>The trajectory</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Relationship between the Interface PSM and the CORBA Objects PSM</td>
<td>25</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Reverse-engineered PSM from CORBA IDL (1)</td>
<td>26</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Reverse-engineered PSM from CORBA IDL (2)</td>
<td>28</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>CORBA PSM with additional associations</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>A CORBA Interface and its Servant</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Class diagram of the implementation</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Simplified sequence diagram of the first binding scenario</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Sequence diagram of the second binding scenario</td>
<td>34</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Full sequence diagram of the first binding scenario</td>
<td>35</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Statechart diagram for VSoDStreamCtrl</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Parallelism by concurrent substates</td>
<td>38</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>One interface and its implementation using delegation</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>Remote and local relationships</td>
<td>40</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Partial class diagram</td>
<td>42</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Flows and Formats</td>
<td>43</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Class diagram of the PIM</td>
<td>44</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Abstract component model of the PIM</td>
<td>45</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Example object diagram</td>
<td>47</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Statechart diagram for StreamCtrl</td>
<td>48</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Statechart diagram for MMDevice</td>
<td>50</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Indirect or direct transformation to WSDL</td>
<td>52</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Graphical representation of one transformed RemoteObject</td>
<td>54</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Class diagram of the Web Services PSM</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Statechart diagram for StreamCtrl</td>
<td>57</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>The Java class used to create the stubs</td>
<td>57</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>One Service and one Port Type implemented using JBuilder</td>
<td>58</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>Abstraction level of the PIM</td>
<td>60</td>
</tr>
<tr>
<td>Figure A.1</td>
<td>Statechart diagram for MMDevice_A in the CORBA PSM</td>
<td>70</td>
</tr>
<tr>
<td>Figure A.2</td>
<td>Statechart diagram for MoviePlayerVDev in the CORBA PSM</td>
<td>70</td>
</tr>
<tr>
<td>Figure A.3</td>
<td>Statechart diagram for MovieReaderVDev in the CORBA PSM</td>
<td>71</td>
</tr>
<tr>
<td>Figure A.4</td>
<td>Statechart diagram for ReceiverStreamEndPoint in the CORBA PSM</td>
<td>71</td>
</tr>
<tr>
<td>Figure A.5</td>
<td>Statechart diagram for TransmitterStreamEndPoint in the CORBA PSM</td>
<td>72</td>
</tr>
<tr>
<td>Figure A.6</td>
<td>Statechart diagram for StreamEndPoint in the PIM</td>
<td>72</td>
</tr>
<tr>
<td>Figure A.7</td>
<td>Statechart diagram for VDev in the PIM</td>
<td>73</td>
</tr>
<tr>
<td>Figure A.8</td>
<td>Statechart diagram for MMDevice in the Web Services PSM</td>
<td>74</td>
</tr>
<tr>
<td>Figure A.9</td>
<td>Statechart diagram for StreamEndPoint in the Web Services PSM</td>
<td>74</td>
</tr>
<tr>
<td>Figure A.10</td>
<td>Statechart diagram for VDev in the Web Services PSM</td>
<td>75</td>
</tr>
</tbody>
</table>
1 Introduction

This chapter presents the motivation, the objectives, and the structure of this thesis.

This chapter is further structured as follows: Section 1.1 gives motivation for middleware independent design of distributed software, Section 1.2 states the objectives of this thesis, Section 1.3 describes the approach that has been followed and Section 1.4 outlines the structure of this thesis by presenting an overview of the chapters.

1.1 Background

Distributed applications are becoming more and more popular. This is enabled by the increase in processing power, the increase in network speed and the decrease in component price.

When constructing distributed applications, often different kinds of programming languages, operating systems and network technologies are involved. This results in the creation of facilities to support interaction between parts of the application for each technology. In order to decrease the effort needed for development, middleware platforms such as CORBA [20], EJB [35] and Web Services [42] are used. Middleware platforms are reusable software infrastructures that provide a collection of application building blocks that facilitate the process of constructing distributed applications [8].

A lot of effort has been invested in building applications using particular middleware technologies, e.g., applications have been built using CORBA. Currently, the trend shifts towards EJB and .NET [16]. Consequently, we have to cope with heterogeneity in middleware platforms. This means that it is not uncommon that multiple middleware technologies are used together. Developers then have to create facilities for each middleware platform to support interaction between middleware technologies. This can be established, for example, by using bridges. Another possibility is to replace parts of the application that use a certain middleware technology so that in the end only one middleware technology is used. There are also other, both technical and commercial, reasons why an application has to be ported from one middleware platform to another. In all cases, it is expensive to rebuild applications for a new middleware platform from scratch.

The Model Driven Architecture (MDA) [24] proposed by the Object Management Group (OMG) aims at solving these problems. Using MDA, Platform Independent Models (PIMs) are used to model an application from a platform independent perspective. From those PIMs Platform Specific Models (PSMs) can be constructed that are specific for a certain platform. Finally these PSMs are used to construct the application. In MDA, a platform is a general term, commonly used to denote a middleware platform.

PIMs capture stable design solutions that are independent of a specific platform. Thus, when a platform changes, PIMs can be reused and only the PSMs have to be recreated. Ordinarily, when a design is platform-specific, platform changes result in a complete or large redesign of
the system. With proper use of MDA, the investment of the design is saved during a change of the underlying technology.

MDA also tries to limit the initial costs of the development. Ideally, the transformation from the PIM to the PSM and the transition from the PSM to the code could be automated. This is only possible when the models are complete and the transformations are completely specified. This would imply that the developer would not use constructs from the PSM anymore, but now creates the software at a higher abstraction level. If full automation can be obtained, the transformation of PIMs to PSMs can be compared to the transition from machine code to assembly or from assembly to higher-level languages made decades ago. However, automated creation of implementations from PIMs is far from reality.

1.2 Goals

It is not always the case that a PIM is available when a platform changes. An example is a legacy system that is developed without the usage of MDA. In those cases MDA can still be useful. From the existing implementation or specification a PSM can be constructed, from that PSM a PIM and from there the top-down approach described earlier can be followed.

To be able to construct a PIM in such a case, we must know what kind of information should be included in a PIM. In this thesis, we identify relevant platform-independent concepts for developing PIMs.

MDA has recently been introduced and there is not much experience with it yet. At this moment it is not very clear to which degree MDA is useful, while much has been promised. We also evaluate the status of MDA and related technologies for its purpose, in particular the use of the Unified Modeling Language (UML) for modeling the structure and behavior of applications along an MDA development trajectory.

1.3 Approach

As a case study we have used an existing framework specified in terms of the CORBA platform, namely the OMG Audio/Video Stream Specification [17]. This framework is chosen because it is a realistic example: the framework is available, documented and implementations already exist. The implementation that is used as an input [38] uses audio- and video-streams and is implemented in Java using the Java Media Framework (JMF) [37].

First, we have created a PSM of the implementation. After that, this CORBA PSM has been transformed into a PIM. This step has been given special attention, because in a top-down MDA approach this step is not included in the trajectory. During that step concepts independent of the CORBA platform have been identified, and these concepts have been used to create the PIM. Finally, a Web Services PSM has been made. After this, an implementation that uses Web Services can be constructed. This last step is outside the scope of this thesis.

The whole trajectory is depicted in Figure 1.1.
We have not automated the transformation between the different models, but we have kept in mind that in future these transformations will be made automatically. We have identified general patterns that can be dealt with automatically, and we have avoided special cases that are only valid in this specific framework as much as possible.

Finally, in the evaluation we analyze the usage of MDA for the transition from one middleware platform to another. Furthermore, we compare the capabilities of MDA for this porting with the capabilities of other possible solutions, such as the transition without the usage of MDA, but with the usage of a specified IDL-to-WSDL mapping [21].

1.4 Structure

The rest of this report has the following outline. Chapter 2 gives an overview of the Model Driven Architecture, introducing technical aspects that are relevant for this thesis. Chapter 3 introduces the Audio/Video Stream Specification, which is used as a case study in this thesis. In addition, the current implementation is discussed, together with the trajectory that has been followed by us.

Chapter 4 describes the Platform Specific Model for the CORBA implementation. Chapter 5 presents the Platform Independent Model in which the platform independent parts are identified. Chapter 6 describes transformation from the PIM into a Web Services PSM. Chapter 7 evaluates the used trajectory. Chapter 8 gives the conclusions of this project.
2 Model Driven Architecture

This chapter introduces the Model Driven Architecture (MDA). The emphasis of this introduction lies on the technical aspects of MDA. The purpose of this chapter is to introduce concepts and terminology that are used in the remainder of this thesis.

First, in Section 2.1 an overview of MDA is given. In Section 2.2 important technologies that are used in MDA are discussed. After that, in Section 2.3 the properties of the different models are given. In Section 2.4 transformations between the different models are discussed. Finally, in Section 2.5 different types of projects that can make use of MDA are discussed.

2.1 Overview

Nowadays, when parts of an application are reused often only code is reused. MDA aims at making reusability at a higher abstraction level possible, by the reusability of design. This reusability of design is enabled by designing the application with the use of models. Another objective of MDA is to increase the development efficiency. MDA aims at achieving this by using tools that automate the creation of new models and code based on existing models.

MDA is not a methodology, nor does it prescribe any methodology. It can be used with many existing methodologies, but some methodologies are currently being developed that are purely targeted at MDA [11]. In order to comply with MDA, a methodology should allow the separation of:
- business rules;
- application logic;
- implementation details.

Business models represent the business processes and business roles without any concern for computational support. In MDA, these models are called the Computational Independent Models (CIMs).

The application logic models the application from a more concrete viewpoint than the business models. The application structure is defined here. However, no implementation details or choices of a platform are made during the creation of the application logic. In MDA, models for the application logic are called the Platform Independent Models (PIMs).

The implementation on a platform consists of the creation of models that reflect the actual implementation. In MDA, these models are called the Platform Specific Models (PSMs). These models can be used when the implementing code is created.

In order to make reusability and automatic creation of models and code possible, the design should be represented in a precise manner. In MDA, this is done by using modeling languages that are well-defined and by well-defined transformations from one model to another. This is
achieved with the usage of metamodels for defining the modeling language explicitly. A metamodel is a precise definition of the constructs and rules needed for creating models [15] and defines the modeling language explicitly in this way. The creation of a new model from an existing model should be done by hand or by automated transformation. A transformation is the systematic replacement of constructs from the old model by new constructs.

2.2 MDA Core and Middleware Technologies

Every MDA trajectory uses one or more technologies that already exist. These technologies play a major role in MDA and they are called the OMG core technologies [24]. In the following sections each core technology is described. In addition to the OMG core technologies, platforms, and in particular middleware platforms, play an important role in MDA. In this thesis, CORBA and Web Services are used and these two middleware platforms are also described.

2.2.1 UML

The Unified Modeling Language (UML) [28] is a standard language for modeling software systems. The UML version 1.5, the current standard, defines nine different types of diagrams. Each of these diagrams can be used to model a system from a certain view. This does not mean that one has to use all the nine types of diagrams when modeling a system. Normally, only a limited number of different types of diagrams is used, depending on the purpose of the model.

UML consists of a collection of notations that can be used to create models. UML is largely process-independent, meaning that you can use it with a number of software engineering processes [2]. This means that the UML does not prescribe a methodology.

In UML, profiles can be used to specialize the language for a specific domain. A profile is defined by a number of tagged values and stereotypes and is a subset of the complete UML specification. Tagged values are name-value pairs that can be used to annotate a construct in a diagram. Stereotypes are model elements that can be used to extend the semantics of a UML construct. A number of profiles have already been designed, such as, for example, the UML profile for CORBA [26]. UML users may also specify their own profiles.

Currently, UML version 2.0 is in the finalization phase. This new version is a major revision of UML 1.5. The number of diagrams is increased to thirteen and a number of other changes are made. In this report we do not make use of UML version 2.0, but we have made our best effort to produce results that are still valid when applied to UML version 2.0.

2.2.2 MOF

The Meta Object Facility (MOF) [23] defines a common metamodel for all the OMG modeling specifications. This metamodel has been used in the specification of the abstract syntax of UML. The MOF version 2.0 and the UML version 2.0 are being aligned to each other, sharing a common subset of core object modeling constructs. Due to this, in the future MOF models can be represented by simplified UML class diagrams that use only UML core elements.

The Extensible Markup Language (XML) Meta-Data Interchange (XMI) [29] is closely related to MOF. The purpose of XMI is to enable exchange and serialization of models that are based on the MOF metamodel. XMI can be used for transporting models between different modeling tools and metadata repositories.
Since XMI is XML-based, other XML technologies such as Extensible Stylesheet Language Transformations (XSLT) [40] or XPath [41] can be applied to XMI. XSLT allows one to perform transformations from one model to another, while XPath allows one to select parts from a model.

2.2.3 CWM

The Common Warehouse Metamodel (CWM) [18] is a standard that is used to model data in an abstract way. CWM compares to data as UML compares to software artifacts. For example, CWM can be used to model a relational database. As with UML, this can be done from different abstraction levels. For example, we can model the data at a low level so that the model is specific for certain database vendor. We can also model the data at a higher level so that the model is usable with all Relational Database Management Systems (RDBMS).

CWM does not play a role in this project, because we concentrate on the structure and dynamic aspects of distributed applications.

2.2.4 Middleware Platforms

In this project two middleware platform are used: CORBA [20] and Web Services [42]. We discuss both of them briefly in this section.

CORBA is an object-middleware that supports method invocation on objects that can be located in different nodes. The first CORBA standard has been created in 1991 by OMG. At this moment the specification is quite mature. The interfaces for the objects are defined in the OMG Interface Definition Language (IDL). Numerous mappings between programming languages and IDL exist. Apart from Remote Method Invocation (RMI), CORBA also includes various services. A CORBA service is a collection of objects and interfaces that support basic functions for using and implementing objects. Examples are the Transaction Services, the Notification Services and the Property Services.

For interoperability CORBA makes use of Internet Inter-ORB Protocol (IIOP) when a TCP/IP network is used. This protocol is blocked by most firewalls, so IIOP does not work well over the public Internet where it is not always possible to change the configuration of the firewalls.

Web Services is a light-weight middleware platform that was created only a few years ago. It makes heavily use of XML. Instead of using binary messages for RMI, plain-text messages are used. This increases the readability of the messages to the human user, but comes with a performance penalty. In Web Services no transport layer is defined, but it is assumed that Web Services use Hypertext Transfer Protocol (HTTP) [7]. This protocol is widely used for transferring web pages and for this reason it is usually not blocked in firewalls.

In contrast to CORBA, Web Services standards are yet maturing. For example, Web Services do not support standardized transactions and notifications by itself. In order to use transactions, one would need to implement the logic in the application or use proprietary extensions.

2.3 Models

A model in the MDA is defined as a description of a system written in a well-defined language. In MDA, the system does not have to be software, however, in most cases it is. An important property of a model is that it uses a language with well-defined syntax and semantics.
A collection of UML diagrams that describes a system forms a model. The meaning of the parts of a UML diagram is defined informally in the UML specification. This has the result that the users of the diagram know how the UML diagram should be interpreted. Other examples of models that can be used are interface definitions, which can be created using languages such as IDL or Web Service Description Language (WSDL). Also source code can be considered as a model according to the definition.

An informal description in English or a sketch with some boxes and arrows with no precise syntax and semantics cannot be considered a model. In both cases the language syntax and the semantics are not well-defined.

In MDA, a number of different types of models are used. Although MDA does not prescribe a methodology, the different types of models can be related to each other in such a way that the total process has the characteristics of a systematic top-down methodology that starts with a business analysis phase and ends up with running code.

During the business analysis phase, Computational Independent Models (CIMs) are created. A CIM models business processes without any computational details. This can be done in UML or in a language that is specialized for modeling business processes. After the CIM Platform Independent Models (PIMs) are created. The type of model that is created after the PIM is a Platform Specific Model (PSM). Finally, the PSMs are used to create the code that implements the system. In this thesis we focus on detailed application development and we use PIMs, PSMs and code and no CIMs.

In Figure 2.1 the dependencies between the different types of models are shown for a top-down methodology. After the PIM multiple PSMs and associated implementations are made, each specific for a certain platform.

![Diagram](Figure 2.1: Dependencies between the CIM, PIM and PSM in a top-down methodology)
2.3.1 Platform Independent Model

In order to be able to refer to the notion of platform independence, we must first define what a platform is. In the MDA, the term platform is used to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component [24]. This definition is general and in practice the term platform can be used for various different concepts.

The term platform has to be used in a more concrete way in order to be useful. Popular usages of the term platform include Operating System (OS), middleware and programming language. When platform is used to denote an OS it is possible that a PIM contains programming language dependent information. In that case the programming language apparently runs on top of multiple OSes and is thus not dependent on the OS. In our project we used the term platform to denote a middleware platform.

By definition, a PIM should not depend on the platform. However, dependence on a platform is not as absolute as it seems. A PIM can depend on the common characteristics of the platforms that are supported by the PSMs. For example, in case the two target middleware platforms of the PIM are both object-middleware, the PIM can use concepts as method invocations. From this point of view the concept of method invocation is independent of the platform. Suppose that at some point in time a third middleware platform needs to be supported and this new middleware is message-middleware. In message-middleware the concept of method invocation is not directly supported. So, when the PIM has all three middleware platforms as potential targets the concept of method invocation is not directly supported in all the platforms, so the PIM is then not platform independent.

This problem can be solved by either introducing a more general concept in the PIM or by creating a mapping from method invocation to something that is directly supported in message-middleware. When the first option is chosen, multiple PIMs can be created. Each PIM can then use its own collection of concepts that is supported by the platforms that the PIM targets. For example, both a PIM that targets object-middleware and a more general PIM can be created. The PIM that targets object-middleware can be created by refining the more general PIM. A drawback of the second approach is that the interaction patterns that are not directly supported by the platform have to be simulated. This is not straightforward and results in loss of traceability and bad performance. In Figure 2.2 the first approach is schematically depicted.

![Figure 2.2: Refinement of PIMs](image-url)
In Figure 2.2 the PIM is refined only once before it is transformed into a PSM. However, multiple refinements may be performed if necessary.

### 2.3.2 Platform Specific Model

A PSM is a model built using concepts from the specific platform. In a top-down trajectory, a PSM is a step forward in creating the implementation. Therefore, a PSM is closer to the implementation on a specific platform, compared with the PIM.

Similarly to the PIM, a PSM can also be refined to create another, more specific, PSM. For example, one can first create a PSM that is independent of the programming language and OS and does not contain any implementation choices concerning these aspects. After that, this PSM can be refined into a PSM in which the implementation choices are made and a language, for example Java, is chosen. Because Java is operating system-independent, in the final PSM the OS still needs to be chosen; the PSM is still independent of the OS. So, in case we define the platform as an OS instead of a middleware technology, this PSM becomes a PIM. Figure 2.3 represents this situation graphically.

![Figure 2.3: PSM becoming a PIM](image)

### 2.4 Transformations

A transformation consists of creating a target model from a source model by rules that describe how one or more constructs from the source model should be replaced by one or more constructs in the target model. A desirable characteristic of a transformation is that it is traceable. This means that it is possible to trace back parts in the target model to their corresponding parts in the source model. This is particularly useful when the target model has to be maintained.

OMG distinguishes four approaches to perform the transformation from a PIM to a PSM [24]:

1. by hand, by replacing constructs by others, without any systematic technique;
2. by hand, using established patterns to convert from the PIM to a particular PSM;
3. by a tool, implementing the established patterns, which produces a skeleton PSM, that is later completed by hand;
4. by a tool that produces the entire PSM.

Each consecutive approach is an evolution of the previous one. In each approach more tools and explicit knowledge are used to perform the transformation. This results in more
automation and less work for the developer. This also means that the transformation rules need to be more complete and better defined. This limits the flexibility of the transformations and introduces more complexity for the transformation rules, which can be a drawback when the transformation is not performed many times.

It is possible that the design decisions and the way the constructs are replaced are hard-coded in the transformation. However, it is better to leave them flexible. The developer then either has to choose between options during the transformation or has to decide between options before the transformation. This second option can be accomplished by introducing markings in the PIM [22]. In this case, certain markings are attached to constructs explaining how the instance of a construct should be translated into a PSM. During the transformation it is then known what choices should be made without having to ask the developer for each construct. With marking, the problem of the lack of flexibility with automated transformation is reduced, but an automated transformation using markings is still less flexible than a manual transformation.

Figure 2.4 shows the transformation from one model to another with the usage of metamodels. First, it is defined how elements in the first metamodel should be transformed into elements in the second metamodel. It is possible that more than one transformation is defined for a certain element. In that case, marks can later be used to specify which transformation should be used. After transformations for each element have been defined, one can transform a model, an instance of the first metamodel, to another model, an instance of the second metamodel, by using the transformation rules defined for each element of the metamodel.

![Figure 2.4: Model Transformation](image)

2.5 MDA Projects

MDA can be used for different types of projects. We briefly describe the following types of projects below:

- Top-down design;
- Reverse-engineering;
- Combined reverse-engineering and top-down design (re-engineering);
- Bridging between platforms.

The type of project that is most commonly described in the available MDA documentation is a pure top-down design of a new system. In this type of project first a CIM is created that models the different processes in the system from a business level. After that, a PIM is created to model the system from a platform independent perspective. This PIM is then transformed into a PSM that is specialized for one specific platform. Finally, the system is implemented on the chosen platform according to the PSM.
When a system is already built and we need exact knowledge about the system design, it is possible to reverse-engineer the application. In a reverse-engineering project, the models are created in the opposite order as in a top-down design project. The code of the system can be transformed into a PSM. From this PIM platform independent concepts can be extracted to create a PIM. From this PIM it is possible to abstract business processes when it is needed to create a CIM.

When a system needs to be ported to another platform it is possible to combine both types of projects described above. First, the application is reverse-engineered, and at some point in the reverse-engineering trajectory the project changes into a top-down design trajectory in order to design the system for the new platform.

The last type of projects differs radically from the rest. In some cases two separate application parts that use different platforms have to cooperate with each other. Sometimes, it is possible to port one system from one platform to another so that both systems use the same platform and can communicate directly. In many cases this is not possible. With a bridge it is possible to automatically translate the communication between two platforms. This requires a PSM for each platform and a general PIM. The bridge intercepts messages for the platform that is not directly supported and transforms them automatically into the appropriate actions that are needed for the unsupported platform.
3 Case Study: Audio/Video Stream

This report uses as a case study an implementation of a video on demand application that makes use of the OMG Audio/Video Stream Specification [17]. This chapter discusses this specification, the implementation and the trajectory that has been followed.

In this chapter, we first introduce the Audio/Video Stream Specification in Section 3.1. After that, in Section 3.2 the existing implementation that is used in the first step of the trajectory is discussed. Finally, in Section 3.3 we discuss the trajectory that has been followed and we look at the scope of the system in the various phases of the trajectory.

3.1 OMG Audio/Video Stream Specification

We start with some terminology used in the specification and the scope of the specification. After that, we discuss the parts of the specification and objects that are used to implement these parts.

3.1.1 Motivation

The streams in the OMG Audio/Video Stream Specification [17] are based on flow interactions, which are sequences of interactions from a producer object to a consumer object, as specified in the Open Distributed Processing Reference Model (ODP-RM) [5].

Operations, one of the types of interactions [6] distinguished by the ODP-RM, are used for the negotiation, the establishing and the controlling of the streams. However, an operation is not suitable for modeling an audio/video stream. For this, flow interactions are used. In Figure 3.1 the different interaction types used in the Audio/Video Stream Specification are shown.

![Figure 3.1: Types of interaction](image)

A stream consists of one or more flows. An example is a stream that consists of one video flow and one audio flow, both flowing from one object (the producer) to another (the consumer). Flows in a stream may have different directions. For example, a telephone stream consists of two audio flows in opposite directions. Figure 3.2 shows a number of examples of streams.
Although the name Audio/Video Stream Specification suggests otherwise, the OMG Audio/Video Stream Specification is not specific to audio and video streams. Applications that use other types of streams can also be constructed using this specification. Examples are applications that use a stream of stock-values or a stream of location information generated with a joystick.

The Audio/Video Stream Specification addresses stream establishment and release, Quality of Service (QoS) adjustment, stream control (start/stop). Furthermore, it allows applications to specify the desired streams. The streaming of data is not prescribed by the Audio/Video Stream Specification. Different protocols can be used for the actual realization of streaming. The specification only makes it possible that the parts responsible for the actual streaming can be configured in such a way that both sides use compatible mechanisms to guarantee interoperability.

### 3.1.2 The Objects

The OMG Audio/Video Stream Specification assumes that each Stream Endpoint consists of three parts: a part that is responsible for the streaming (Stream Adaptor), a part responsible for the communication with a centralized controller for the configuration (Stream Interface Control) and a part that actually does something with the stream (Flow Data Endpoint).

The Stream Adaptor is the part that handles the physical streaming. For example, this could be an object that can receive an MPEG-4 flow over Real-Time Transport Protocol (RTP) [13]. This Stream Adaptor is configured by the Stream Interface Control part of the endpoint and a centralized Stream Control and Management object. During this configuration attributes such as the format of the flow, the frame rate, the data rate, compression options, encryption options, the underlying network technology, etc. must be agreed upon. CORBA is used to support interactions between the Stream Interface Control part and the Stream Control and Management object. The Flow Data Endpoint is the part in which the data that is streamed is produced or used. For example, this can be an object that reads an MPEG-4 file from disk and passes it to the Stream Adaptor.

In Figure 3.3 two Stream Endpoints and the connections between them are shown. The bold lines represent the interfaces specified in the specification.
Since the specification only prescribes functionality for stream management, the Flow Data Endpoint and the Stream Adaptor parts and their interfaces are not specified. Each Stream Interface Control consists of three CORBA Objects [20]. Figure 3.4 shows these objects, together with the object that is the centralized controller.

An MMDevice object abstracts a multimedia device with certain capabilities. An example could be an object that abstracts a microphone. An MMDevice object acts as a factory for VDev objects and StreamEndPoint objects.

A VDev object represents a virtual device, which is the computational representation of an MMDevice. For each stream connected to the MMDevice a separate VDev object is created.

A StreamEndPoint object controls the flows of the stream. For example, this object can start and stop the flows.

The StreamCtrl object handles the overall configuration. It can be situated at the same node as one of the Stream Endpoints, but it can also be situated at an entirely different node.

The specification consists of two profiles, namely the light profile and the full profile. The light profile is a subset of the full profile and is used in light weight implementations where the number of method invocations should be kept to a minimum. The full profile allows individual flows to be controlled. This results in finer control of the streams.
3.2 Initial Implementation

For our case study, we start with an existing implementation of a simple video-on-demand application [38] that uses the Audio/Video Stream Specification.

Since the Audio/Video Stream Specification does not address the streaming itself, nor the usage of the data that is streamed, the specification only defines a framework, which should be embedded in a complete application. Applications usually do not use streams in a general way, but they use a more specialized subtype of streams, such as audio/video streams. It is then possible to use or to develop an implementation of the specification that is specialized for those streams.

We used an implementation of a simple Video Streaming on Demand (VSoD) application. The system consists of a server and a client. The server has video files available at disk and sends information about those files to the client. The client thus knows which video files are available from the server. The client requests one of these files and the server reads the specified video file from disk and streams it to the client. The client receives the video and plays it on screen so that the user can watch the video.

The implementation has been developed in the scope of the Application of Middleware in Services for Telematics (AMIDST) project [1] and is reported in [38]. The application implements the OMG Audio/Video Stream Specification and a VSoD client and server, which are not part of the specification. The system only uses audio and video streams, so that the implementation of the specification is specialized for those kinds of streams.

Figure 3.5 depicts the system is schematically and is based on Figure 3.3. The objects of the application that have not been specified in the standard are in the box with number 1. An example of a piece of the system that can be found here is the user interface. The box with number 2 is the part of the application that has been specified by the specification. For example, a general version of a StreamEndPoint can be found here. The parts with number 3 are the parts of the application that include specializations of the specification to make it useable for VSoD. A StreamEndPoint that is specialized for video-on-demand streaming can be found here.

Figure 3.5: Construction of the system

The application only supports streams with two flows, namely one video flow and one audio flow, both originating from the server and terminating at the client. The server reads a file with the video and the audio from disk. The client plays the received audio and video. The way the server should read the file and the way how the client should play the video are not specified in the standard. In the implementation these parts are implemented using the Java...
Media Framework (JMF) [37]. RTP [34] over User Datagram Protocol (UDP) is used as transport protocol. Support for RTP over UDP is already built into JMF and JMF has thus been used for this.

This implementation only implements the light profile. Furthermore, the implementation does not support multicasting or Quality of Service (QoS).

3.3 The Trajectory

3.3.1 The Phases

The implementation described in Section 3.2 is used as a starting point for our MDA trajectory.

The first step of the trajectory has been the creation of the model of the existing application, the PSM. More specifically, this model is made of the parts of the application that are related to the Audio/Video Stream Specification.

After that, we have created a PIM from that PSM which does not contain platform-specific information. We have tried to generalize this PIM as much as possible both with respect to the general stream configuration capabilities defined in the Audio/Video Stream Specification and with respect to the degree of platform-independence. For example, the implementation uses only one audio and one video stream and both streams originate in the server and are terminated in the client. The implementation of the specification only supports this configuration.

The PIM has been transformed into a Web Services PSM. This PSM contains the same kind of information as the CORBA PSM.

After the Web Services PSM an implementation similar to the existing implementation can be constructed. The new implementation does not use CORBA as the middleware platform, but Web Services. When the programming language remains Java, some CORBA-independent parts of the original implementation can be reused. In Figure 3.6 the whole trajectory is shown graphically.

![Figure 3.6: The trajectory](image-url)
3.3.2 The Scope

With the construction of the CORBA PSM we have created a model of the current implementation. Because we are mainly interested in the Audio/Video Stream Specification only parts 2 and 3 from Figure 3.5 have been modeled. This means that our CORBA PSM has a similar scope as the implementation of the complete system. This PSM supports all kinds of audio/video streams, not only those required for the VSoD implementation. However, our CORBA PSM is more specific than the OMG Audio/Video Stream Specification. In the specification, the types of streams are not fixed. Furthermore, the specification also supports multicasting and parts of the specification are specific for the full profile. In the implementation placeholders are defined for multicast, the internal flow control protocol and QoS. Because the implementation contains placeholders and the PSM represents the implementation as much as possible with respect to the objects and the operations, the CORBA PSM also contains placeholders. Table 3.1 shows the differences between the Audio/Video Stream Specification, the CORBA implementation and the CORBA PSM.

<table>
<thead>
<tr>
<th>A/V Stream Specification</th>
<th>CORBA Implementation</th>
<th>CORBA PSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>- multicast / unicast</td>
<td>- only unicast, placeholders for multicast</td>
<td>- only unicast, placeholders for multicast</td>
</tr>
<tr>
<td>- light / full profile</td>
<td>- only light profile</td>
<td>- only light profile</td>
</tr>
<tr>
<td>- internal flow control</td>
<td>- placeholders only</td>
<td>- placeholders only</td>
</tr>
<tr>
<td>protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- QoS support</td>
<td>- placeholders only</td>
<td>- placeholders only</td>
</tr>
<tr>
<td>- generic stream types</td>
<td>- video on demand</td>
<td>- A/V streams</td>
</tr>
</tbody>
</table>

In the PIM we did not have to consider the implementation anymore, so the placeholders that are modeled in the CORBA PSM did not have to be modeled anymore. Furthermore, in the Web Services Implementation we could specialize the type of streams again to video on demand. In Table 3.2 the same properties as in Table 3.1 are listed for the PIM, the Web Services PSM and the Web Services Implementation.

<table>
<thead>
<tr>
<th>PIM</th>
<th>Web Services PSM</th>
<th>Web Services Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- only unicast</td>
<td>- only unicast</td>
<td>- only unicast</td>
</tr>
<tr>
<td>- only light profile</td>
<td>- only light profile</td>
<td>- only light profile</td>
</tr>
<tr>
<td>- no internal flow control protocol</td>
<td>- no internal flow control protocol</td>
<td>- no internal flow control protocol</td>
</tr>
<tr>
<td>- no QoS support</td>
<td>- no QoS support</td>
<td>- no QoS support</td>
</tr>
<tr>
<td>- A/V streams</td>
<td>- A/V streams</td>
<td>- video on demand</td>
</tr>
</tbody>
</table>

For the PIM, the Web Services PSM and the Web Services Implementation, the terms light and full profile cannot refer anymore to the terms in the Audio/Video Stream Specification, because the PIM, the Web Services PSM and the Web Services Implementation are not directly dependent on the CORBA specification. However, the terms are still used to denote the functionality of the light and the full profile from the specification.
This chapter describes the first step in our development trajectory, namely the creation of the CORBA PSM. We do not give the complete models in this chapter, but we concentrate on the design decisions made. This is because large parts of the models are similar to each other.

We start in Section 4.1 with a discussion about the different PSMs that have been created. In Section 4.2, we discuss the model that corresponds to the CORBA interfaces defined in the Audio/Video Stream Specification. The model that corresponds to the realization of those CORBA interfaces is discussed in Sections 4.3 and 4.4. Section 4.3 models the static view of the system: the static structure of the system with class diagrams. Section 4.4 models the dynamic view of the system: this view defines how the objects behave. Finally, in Section 4.5 we discuss the Realization PSM in which the implementation of the CORBA Objects is modeled.

4.1 Overview

We define three different PSMs in this step. Initially, we consider the Interface PSM. In this PSM only models the CORBA interfaces, so there is no dependence on the programming language or implementation choices. After that, the Interface PSM is further refined and the CORBA Objects and their interactions are considered. This results in the CORBA Objects PSM, which includes a static view and a dynamic view. Finally, the Realization PSM models the implementation of the CORBA Objects for the programming language Java. In Figure 4.1 the relationship between the Interface PSM, the CORBA Objects PSM and the Realization PSM is illustrated.
4.2 Interface PSM

In this PSM we only model the system by using what is specified in the IDL specification. The UML profile for CORBA [26] specified by the OMG has been used to produce the Interface PSM. However, at the end of this section, we extend one diagram with self-defined associations.

The UML profile for CORBA provides a mapping from OMG IDL [19] to UML. An IDL file consists of a number of declarations. For most of these declarations a corresponding graphical UML representation has been defined. For example, for a module in IDL a package with the stereotype `CORBAModule` is used. Table 4.1 gives the IDL elements that are used in the Audio/Video Stream Specification, together with their corresponding UML Profile stereotype. The complete profile can be found in [26].

<table>
<thead>
<tr>
<th>IDL element</th>
<th>UML profile stereotype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>Package with stereotype CORBAModule</td>
</tr>
<tr>
<td>Interface</td>
<td>Class with stereotype CORBAInterface, the operations of the interface are defined as operations in the class</td>
</tr>
<tr>
<td>Exception</td>
<td>Class with stereotype CORBAException, the parameters of the exception are defined as attributes of the class</td>
</tr>
<tr>
<td>IDL built-in types</td>
<td>Class with stereotype CORBAPrimitive</td>
</tr>
<tr>
<td>Typedef</td>
<td>Class with stereotype CORBATypedef, with association to the original type</td>
</tr>
<tr>
<td>Sequence</td>
<td>Class with stereotype CORBASequence, with aggregations to the element type</td>
</tr>
<tr>
<td>Struct</td>
<td>Class with stereotype CORBAStruct, where the members of the struct are shown either as an attribute or as a named aggregations to the type</td>
</tr>
<tr>
<td>Enum</td>
<td>Class with stereotype CORBAEnum, where the members of the enumeration are listed as attributes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
In order to start the generation of the PSM we have used the Reverse Engineering function of Rational Rose. This results in a UML diagram that needs just a few manual adjustments. First of all, the CORBAPrimitive-Classes were not generated and had to be added by hand. Furthermore, instead of CORBAInterface-Classes normal Interface-classes were generated. The relationship between the structs and sequences and their member elements were shown as normal associations instead of aggregations. Finally, the member QoSParams of the struct QoS was created as an attribute of QoS. Because QoSParams is a user-defined type it should have been created as a separate class and QoS should have an association to it. After these adjustments, the final result is the class diagram shown in Figure 4.2 and Figure 4.3.

Figure 4.2 shows the CORBA interfaces. Most of the interfaces are already discussed in Section 3.1.2, except for McastConfigIf and Negotiator, which are used for multicasting and negotiation respectively, and interface PropertySet. The latter interface is defined in the Property Service [25], which is a CORBA service. All the other interfaces in the Audio/Video Stream Specification inherit from the PropertySet interface, because the PropertySet interface defines general operations to define, modify and inspect property values.

Figure 4.2: Reverse-engineered PSM from CORBA IDL (1)
The rest of the model is shown in Figure 4.3. At the top of the figure a number of exceptions are shown. A number of them have a string as attribute, which is an exception parameter. Operations in the interfaces can raise these exceptions. Tagged values of operations are used to define which operations can raise which exceptions in the UML Profile for CORBA. Unfortunately, tagged values are not shown in a diagram produced using Rational Rose, so they have been omitted in the figures presented here.

At the bottom part of Figure 4.3 the user defined types are shown. Some types are related to each other by associations. The types protocolSpec and flowSpec are both sequences. Because they are both strings, they are associated to the CORBAPrimitive string. The type streamQoS is a sequence of QoS. String is a built-in type, and thus has the stereotype CORBAPrimitive.

Two classes have the stereotype CORBATypedef, namely key and streamEvent, which means that they rename another type. The class key renameS octet, so that in the model key is associated to the octets built-in type. Since streamEvent renameS Property, in the model it is associated to the Property class. Property is a struct defined in COSPropertyService.

Two enumerations have been defined in the IDL specification, namely flowState and dirType. These enumerations are represented by using the stereotype CORBAEnum. The member elements of the enumeration are listed as attributes in the class. For example, the enumeration flowState has the member elements stopped, started and dead.

Finally, three structs are shown in Figure 4.3. SFPStatus only has members of built-in types. These members are shown as attributes of the class. The struct flowStatus has a member of a built-in type and other members of user-defined types. The member of the built-in type flowName, with the type string, is shown as an attribute. The other members are shown as associations to the type of the member. The name of the association is the name of the member. For example, flowStatus further consists of directionality with type dirType, status with the type flowState, theFormat with type SFPStatus and theQoS with type QoS.
In Figure 4.2 the interfaces and the hierarchical relationships between them are shown, associations between interfaces have been omitted. We have defined two types of associations, namely a use-association and a create-association. These associations are used in Figure 4.4.

A use-association is created from a source interface to a target interface when the realizing class of the source interface must make use of a realization of the target interface. These associations can be taken from the IDL interface file. It is absolutely necessary for a realization of an interface to use another realization of an interface when the name of the target interface is found as a parameter of an operation in the source interface.

A create-association is created from a source interface to a target interface when the realizing class of the source interface must create a realization of the target interface.
Figure 4.4 shows the CORBA PSM containing the classes derived from the IDL specification, but with additional associations discussed above.

4.3 CORBA Objects PSM Static View

The interfaces as described in the Interface PSM, only specify the signature of the operations that should be implemented. They do not specify the behavior of the realizing CORBA Objects. In order to capture this type of information, the CORBA Objects PSM models the CORBA Objects that implement the interfaces.

When a CORBA interface is implemented, a CORBA servant is used for that purpose. We now want to specify the associations and the behavior of the servants that implement the interfaces.
Each interface is realized by a servant. We have chosen to represent this by the fragment shown in Figure 4.5. The stereotypes CORBA\textit{Servant} and CORBA\textit{Interface} are used to denote a servant and an interface, respectively. The relationship between the interface and the servant is a realize-relationship.

![Figure 4.5: A CORBA Interface and its Servant](image)

The OMG Audio/Video Stream Specification defines two parties, A and B, to make it possible to define the direction of the flows properly. The implementation of the different parties can be specialized for that party. For example, an implementation of the MMDevice interface of party A does not have to implement the operation Create\textsubscript{B}. For each CORBA interface a general class has been defined that includes only general code that is applicable to both parties. Using inheritance two classes are created, one for each party, which implement the code that is applicable only for a specific party. This means that the number of classes in the implementation is much larger than the number of CORBA interfaces. However, there is a one-to-one mapping between the class diagram of the CORBA interfaces and the class diagram of the implementation. This class diagram of the implementation is depicted in Figure 4.6.

In addition to the CORBA\textit{Interface} and CORBA\textit{Servant}, the Implementation stereotype is used to denote that the class is a normal implementation class implemented in the programming language that is used for the implementation.
4.4 CORBA Objects PSM Dynamic View

Interfaces just specify which operations should be implemented, but they do not specify how that should be done or what result an operation invocation should have. When we talk about
behavior here, we talk about how the implementation of the interface, the servant, should behave.

4.4.1 Sequence Diagrams

We define the dynamic view by creating a number of sequence diagrams. Each of these diagrams describes a different part of the total behavior. The part in which most things happen is the establishment of a stream. According to the standard, this can be done in many different ways. Two scenarios modeled in the sequence diagrams are listed in Table 4.2.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The MMDevice objects and the StreamCtrl object are created during initialization, before the scenario is started. The establishment is started with an operation invocation on the StreamCtrl interface.</td>
</tr>
<tr>
<td>2</td>
<td>Only the MMDevice objects exist at the start of this scenario. The StreamCtrl object is created after the establishment is started with an operation invocation on an MMDevice.</td>
</tr>
</tbody>
</table>

In Scenario 1, two MMDevice objects and one StreamCtrl object initially exist. At the start, the StreamCtrl delegates the creation of the VDev and the StreamEndPoints objects to the MMDevice objects. After these objects are created the VDevs are related to each other. When this happens the VDevs have to be configured so that the binding between the parties is supported by both parties. Finally, the relating StreamEndPoints are connected to each other.

Figure 4.7 shows a simplified view of this scenario. In this scenario all the operation invocations that are related to the interface are modeled. The objects also store and access their own properties and attributes, but we have omitted this in this diagram.
Scenario 2 is similar to scenario 1. The main difference is that the MMDevice object in scenario 2 creates the StreamCtrl objects. After this is done scenarios 1 and 2 become identical. This is why only the beginning of this scenario is modeled in Figure 4.8.
The realization of the IDL interfaces uses properties to hold information. To specify this, we have created a more detailed version of Figure 4.7. In Figure 4.9 we also show how the CORBA Property Service is used. Two operations, `get_property_value` and `define_property`, are used to retrieve and to store the values of properties respectively. The values of the arguments are shown in comment boxes connected to the operations. In case an operation is called more than once in succession, only one operation invocation is modeled, but more values are shown in the comment box.
Figure 4.9: Full sequence diagram of the first binding scenario

### 4.4.2 Statechart Diagrams

Sequence diagrams just show parts of the behavior, but usually one wants to model the system in a complete way, in terms of all possible execution alternatives. It is possible to define a number of alternatives in a sequence diagram, but this is not very practical, because then the models become very unclear. In practice, in a sequence diagram only one possible sequence of actions can be modeled. In Figure 4.9 and Figure 4.8 the VDev and StreamEndPoint objects are created after the bind_devs or bind operation is invoked. It would also be possible to create these objects before the operation invocation. The objects VDev and StreamEndPoint would behave in the same manner. It is possible to define more scenarios in which these objects are created beforehand, but this is not desirable, because the number of diagrams will become very large.

Since we would like to preserve a maximum level of implementation freedom, we model the dynamic view in such a way that the implementing developer can make more decisions on the
implementation. We define a statechart diagram for each object and omit information about when an object should be created. A statechart defines the behavior of an object in terms of its states. For each state, a statechart defines the actions that occur after a certain operation is called and the new state of the object after performing these actions.

In Figure 4.10 the statechart for the VSoDStreamCtrl is depicted. The diagram consists of states, transitions and comment boxes. In general, states are not shown explicitly in code. To obtain the statechart diagram we have used the variables of the code and the informal text of the specification. Because the informal text can be interpreted in many different ways, the statechart diagram shown is just one of the many possible diagrams.

A transition between two states consists of an event before the `/`-symbol and an action or an action sequence after it. The name of the event is the name of the operation that is invoked and triggers the event. The concrete syntax of the actions is not specified in the UML specification; we have used the same notation as in [30]. We have only used call actions and create actions. With a call action the name of the operation that is called is used. When an action corresponds to calling a method on another object, the name of the object is prefixed to the action. A create action consists of the keyword `new` followed by the name of the class of which a new instance should be created. More than one action can be defined. In such an action sequence, semicolons denote sequencing. Comment boxes are used in the diagram to assign pre- and postconditions to transitions. These pre- and postconditions are written in the Object Constraint Language (OCL), specified in the UML specification [28].

The statechart also shows three important states: Initialized, SEPs and VDevs created, and the composite state Stream Established. The transitions between these states have comment boxes to define pre- and postconditions for the operations. The object goes only from the Initialized state directly to the Stream Established state when the SEPs and VDevs exist. When this is not the case, the defer statement in the Initialized state makes sure that the treatment of the bind operation call is postponed. The transition from the state Initialized to the state SEPs and VDevs created only happens when the SEPs and VDevs do not yet exist. The postcondition linked to bind_devs ensures that after that operation invocation the SEPs and VDevs do exist. The bind operation that results in the transition from SEPs and VDevs created to Stream Established has the same pre- and postconditions attached as the bind operation that
results in the transition from **Initialized** to **Stream Established**. Again, this operation can only be called when the **SEPs** and **VDevs** exist; the postcondition from **bind_devs** ensures that this is always the case.

When the **bind_devs** operation is called, the action states that first **sep_a.create_A** and **sep_b.create_B** have to be performed. These operation calls result in the creation of the **SEPs** and the **VDevs**. After that, the **bind** operation is called. The invocation of the operation **bind** results first in the invocation of the operation **vdev_a.set_peer**, then the invocation of the operation **vdev_b.set_peer** and finally the invocation of the operation **s_a.connect**.

The composite state **Stream Established** consists of two sub states, **Stream stopped** and **Stream started**. After the stream has been established, the object is in the **Stream stopped** state. When the operation **start** is called the stream is started by calling the **start** operation first on **sep_a** and after that, on **sep_b**. When the stream is started it can be stopped with the operation **stop**, caused the operation **stop** to be called on **sep_a** and **sep_b**.

When the object is in the state **Stream started** a new call on **start** is ignored. The same holds for the **Stream stopped** state, in which an invocation of **stop** is ignored. In the statechart diagram we represent explicitly that a certain operation is ignored. This is not necessary, but we have decided to do this because in this way the consequences of invoking an operation are explicitly modeled for each operation.

When the object is in the state **Stream Established** and the operation **destroy** is called, the object returns to the **Stream not Established** state, and the operation **disconnect** is called on the **sep_a** and **sep_b** objects.

A number of error-states are shown. Some operations bring the object to an error-state, independent of the current state. These are operations related to QoS, Flow Control and the full profile, which are not modelled. Operations that need an established binding may also result in error states when the binding is not yet established. When the object is in an error-state an exception is thrown and the object returns to its previous state. This is due to the history state, which is drawn in the global **OK** state in which the object always resides when no error occurs. The history state remembers the last sub-state and takes care that when the state returns to the composite **OK** state, the state is set to that sub-state.

### 4.4.3 Discussion

For each of the other objects a similar statechart diagram is created. Because they are created in the same way as the statechart diagram for the **VsoDStreamCtrl** they are not discussed here.

Unfortunately, the concrete syntax for the actions that we have used and the abstract action semantics defined in UML does not support parallelism. In some cases it does not matter in which order parts of an action are executed. For example, after the operation **bind_devs** is invoked, the operations **create_A** and **create_B** are called. Although Figure 4.10 specifies that the operations are called in that specific order, it does not matter in which order these operations are called. If the syntax would support parallelism, we could replace the line

\`
\text{bind\_devs / m\_a.create\_A, m\_b.create\_B, bind}
\`  

by something like

\`
\text{bind\_devs / (m\_a.create\_A | m\_b.create\_B), bind}
\`

to define that first the operations **create_A** and **create_B** are called in any order and after both calls return the operation **bind** is called.

Another option would be to use activity diagrams instead of statechart diagrams. In an activity diagram one can use concurrent substates to model parallelism. For example, the line

\`
\text{bind\_devs / m\_a.create\_A, m\_b.create\_B, bind}
\`  

can be replaced by the fragment shown in Figure 4.11. However, a drawback of this approach is that the notation is not compact and distracts from the real essence of the diagram.
Creating objects

bind_devs

/ m_a.create_A

/ m_b.create_B

Figure 4.11: Parallelism by concurrent substates

The combination of the sequence diagrams and the statechart diagrams can be used to verify whether the interactions expressed by a sequence diagram can be realized by the set of statechart diagrams. In this way one can check whether the dynamic view of the system is consistent. For each object a token is placed at the initial state of the statechart diagram. Then the actions from the sequence diagram are handled one by one. For each action, it must be verified that the step is possible according to the sequence diagram and when this is the case the token must be moved to the specified new state. In case the step is not possible according to the sequence diagram the verification process can be aborted, because then the sequence diagram cannot be realized by the set of statechart diagrams.

This verification can be done by a tool such as, for example, Hugo [33]. This has the benefit that the verification can be done in a relatively short time and with no manual work. This encourages the developer to verify the consistency of the dynamic view as often as possible, possibly even after every change. In this way, inconsistencies are spotted and handled early during the development. A drawback of using a tool is that you have to invest time, and possibly money, to get the tool, to install and configure it, and to learn how to use it. Because we only need to verify the consistency once, we have done this manually.

4.5 Realization PSM

The Realization PSM specifies the objects that realize the CORBA Objects. This PSM has more details than the CORBA Objects PSM, since the details of the implementation of the CORBA Objects are modeled.

For the purpose of porting the application from one middleware platform to another the realization diagram in the PSM of the original platform does not play a large role. It represents implementation choices that are on a lower abstraction level than the Interface PSM and the CORBA Objects PSM. The Realization PSM is included for completeness. A PIM might be used for more than just the porting, for example, it might be used for documentation. In that case the Realization PSM is an important model.

The objects that handle requests to CORBA Objects, the so-called servants, are not the only constructs that we need when we model the implementation. It is also important to define how an implementation realizes its CORBA interface [10]. Two possible implementation techniques that can be used for this are inheritance and delegation. Because delegation is used in the implementations, inheritance is not discussed here.

When delegation is used, each CORBA interface realized results in eight Java classes and interfaces for the implementation [4]. For a CORBA interface called I, the IHolder-class is created to accommodate the passing of CORBA objects as parameters. The IHelper-class implements a collection of methods for narrowing the object references, marshalling and
generalization to Any types. The IImpl-class is the servant that implements the operations of the CORBA object. This class is the only class that needs to be adjusted by hand. The IStub-class implements the stub that can be used by clients to invoke operations on the CORBA Object. The interface I and the IOperations-Interface are used to be able to refer to the CORBA Object and define the operations that need to be supported. Finally, the IPOA- and IPOATie-classes are used to delegate the incoming invocations of operations to the servant.

For our nine CORBA interfaces this would result in at least 72 classes to be drawn in one model. Since this is not practical, we first give the complete model for an interface and its implementation in Figure 4.12, and, later, we introduce a UML tagged value \texttt{realization = delegation} (as in [10]) that can be used to specify the connection between an interface and the implementing class. When this tagged value is set, the interface is implemented in the way depicted in Figure 4.12. The stereotypes \texttt{JavaInterface} and \texttt{Implementation} are taken from the UML Profile for EJB [12].

![Figure 4.12: One interface and its implementation using delegation](image)

When the structure in Figure 4.12 is collapsed to a class and an interface (such as in Figure 4.13), the interface shown is not a Java interface, but a CORBA interface. This CORBA interface has the same name as the Java interface and is also implemented by the Java interface.

Between the CORBA interfaces and the implementing classes, different kinds of associations can be made. In Figure 4.13 the four different possibilities are shown.
In the case of association R1, the servant uses a class for the implementation and has a local relationship with a class. In this case no middleware is used in their interaction, i.e., the instances of these classes interact directly. Objects of these classes need to reside on the same node and the relationship must be created using constructs in the implementing language.

In the case of association R2, the servant uses another CORBA object. The reference refers to the interface of the CORBA object, and the CORBA object can even be located at a different node and implemented using a different programming language than the servant implementing the CORBA Object.

A CORBA interface cannot have an association with an implementing class except with the associated servant. This is because the interface is programming language-independent and a servant is programmed in a certain language. For this reason, the third kind of relationship cannot exist. This is not the case for interfaces in programming languages. For example, a Java interface can have an association with another class. In this case both the interface and the referenced class are programmed in the same language and the relationship is local.

Finally, association R4 is an association between two CORBA interfaces. This is possible, because both interfaces are language independent. When a CORBA interface is associated to another CORBA interface, the referencing interface has one or more operations that have the referenced interface as a type of a parameter.
5 Platform Independent Model

This chapter describes the development of the PIM, which corresponds to the second step in our trajectory. In this chapter we discuss the important modeling decisions made and the relationship with the CORBA PSM.

The PIM is created based on the CORBA PSM. This means that we have generated the PIM by transforming the CORBA PSM diagrams. In Section 5.1 the static view is discussed. This view consists of a class diagram and an abstract component diagram in which the classes defined in this section are grouped together. In Section 5.2 the dynamic view is discussed. We also introduce an action language that is used in our statechart diagrams.

5.1 Static View

5.1.1 The Classes

In this section we create a class diagram based on the static view of the CORBA Objects PSM. All the classes in the newly created diagram have the stereotype remoteObject to denote that the instances are remote objects that are implemented using a middleware technology and that each of them can be located at a different node.

In the implementation and in the CORBA PSM, inheritance is used to discern between the differences in implementation of the different parties. At a higher abstraction level, we decide to ignore these differences. This makes the PIM applicable for PSMs in which the distinction between different parties is not important. Ignoring the differences means that, for example, the classes StreamEndPoint, StreamEndPoint_A and StreamEndPoint_B can be combined into one remoteObject. The class StreamEndPoint now supports everything that both StreamEndPoint_A and StreamEndPoint_B support, so in every place where StreamEndPoint_A has been used this can be replaced by StreamEndPoint.

The same holds for the inheritance that is used for the specialization of the Audio/Video Stream Specification for the implementation. The classes TransmitterStreamEndPoint and ReceiverStreamEndPoint thus disappear in the PIM and are combined in the class StreamEndPoint.

It is only possible to combine the classes when the resulting class supports the behavior of all the classes that are combined. For example, MMDevice_A is specialized in the creation of VDev and StreamEndPoint objects for party A and cannot create objects for party B, and MMDevice_B is specialized in the creation of VDev and StreamEndPoint objects for party B and cannot create objects for party A. When these two classes are combined the class MMDevice has to be able to create VDev and StreamEndPoint objects for both party A as party B.
The main advantage of combining the classes is that the number of classes decreases. This results in a clear class diagram. The main drawback is that the classes become more complex, since they must support the behavior of both parties.

The removal of the inheritance decreases the number of classes to just four. These two steps result in the class diagram fragment shown in Figure 5.1.

![Class diagram fragment](image)

**Figure 5.1: Partial class diagram**

A ScreamCtrl object is connected to at most two MMDevice objects (R1), which function as source and sink for the stream. An MMDevice object supports multiple simultaneous streams, for each stream another connection to a StreamCtrl object is used. For each stream (R2) a VDev object and a StreamEndPoint object are created. These two objects can be connected to each other (R3).

### 5.1.2 Properties

In the Audio/Video Stream Specification and in the VSoD implementation, properties are used to attach information to objects in a flexible way. In the PSM these properties are potentially distributed and are managed by a standardized interface, but this is irrelevant from an abstract view.

Properties are used in the Audio/Video Stream Specification for two purposes:
- to hold data about an object;
- to refer to another object.

In the first case the property can be replaced by a class attribute. In the second case the property can be discarded if we model the reference as a navigable association in the UML class diagram.

Therefore, for each property we have to determine its purpose. In Table 5.1 the properties from the sequence diagram of the CORBA PSM shown in Figure 4.9 are listed.
<table>
<thead>
<tr>
<th>Class</th>
<th>Property</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamCtrl</td>
<td>A_Parties</td>
<td>Reference</td>
</tr>
<tr>
<td>StreamCtrl</td>
<td>B_Parties</td>
<td>Reference</td>
</tr>
<tr>
<td>VDev</td>
<td>Flows</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>&lt;flowName&gt;_dir</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>&lt;flowName&gt;_availableFormats</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>&lt;flowName&gt;_currFormat</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>Related_StreamEndPoint</td>
<td>Reference</td>
</tr>
<tr>
<td></td>
<td>Related_MMDevice</td>
<td>Reference</td>
</tr>
<tr>
<td>StreamEndPoint</td>
<td>Flows</td>
<td>Data</td>
</tr>
<tr>
<td>StreamEndPoint</td>
<td>&lt;flowName&gt;_dir</td>
<td>Data</td>
</tr>
<tr>
<td>StreamEndPoint</td>
<td>&lt;flowName&gt;_currFormat</td>
<td>Data</td>
</tr>
<tr>
<td>StreamEndPoint</td>
<td>Related_VDev</td>
<td>Reference</td>
</tr>
<tr>
<td>StreamEndPoint</td>
<td>Related_StreamCtrl</td>
<td>Reference</td>
</tr>
<tr>
<td></td>
<td>AvailableProtocols</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>PeerAdapter</td>
<td>Reference</td>
</tr>
</tbody>
</table>

The Flows property and the <flowName>_dir, <flowName>_availableFormats and <flowName>_currFormat properties are implicitly related to each other. These relationships can be modeled in the way depicted in Figure 5.2.

![Figure 5.2: Flows and Formats](image)

The Flows property defines the name of the flow. This name is used in the Audio/Video Stream Specification to relate the rest of the properties to that particular flow. For example, when we have a flow that is called video, properties such as video_dir and video_availableFormats are used to define properties about the flow video. The <flowName>_dir specifies the direction of the flow. The other two properties define the format of the flow. The <flowName>_availableFormats is a collection of formats the flow supports and <flowName>_currFormat is the format that is currently selected. Again, the name is used as an identifier.

We could transform the properties by making all of them attributes of their corresponding classes. In this case, for example, the attributes Flows, video1_dir, videol_availableFormats, videol_currFormat, audio1_dir, audio1_availableFormats and audio1_currFormat would be attributes of the class VDev. This option has two major drawbacks. First of all, the number and the names of the flows would be fixed, for two flows, video1 and audio1, as in the PSM. In case we want to change the flows we also have to change the model. Furthermore,
structure and the relationship between the flows and the formats given in Figure 5.2 would be lost. For these reasons the classes from Figure 5.2 have been added to the class diagram instead of the explicit attributes.

There is still one property left that holds data, which is `AvailableProtocols` at the `StreamEndPoint` class. This property has been modeled as an attribute of `StreamEndPoint` in the PIM, because it holds a string as data.

Operations used to manipulate properties, such as `get_property_value` and `set_property` are not necessary anymore, because properties are not used in the PIM. Operations that are related to QoS or multicasting and are modeled in the PSM, because a placeholder has been created in the implementation, can be ignored as well. The rest of the operations can be mapped directly to the operations of the PIM.

After the classes have been combined, the properties have been transformed into attributes and classes, and the superfluous operations have been deleted, the total class diagram of the PIM has been obtained. This final class diagram is shown in Figure 5.3.

![Figure 5.3: Class diagram of the PIM](image)

5.1.3 Abstract Components

Since CORBA version 2.x does not have a concept of components, the Audio/Video Stream Specification and the VSOD implementation do not use components. For this reason the CORBA PSM does not include a component diagram. Components in UML version 1.5 are units of deployment. Since the PIM models the system in a distribution transparent way we cannot use deployable components on this level, since these are platform dependent.

However, it is useful to be able to depict that a number of interfaces are closely related to each other. For this, we use abstract components, similar to the ProcessComponents used in
the Component Collaboration Architecture (CCA) and defined in Enterprise Distributed Object Computing (EDOC) [27].

The remoteObjects used in the Class diagram are transformed into two entities here: an interface and a port. The interface is only used for the interface definition. The port related to the interface implements the interface. Another approach was to directly relate interfaces to the abstract component. This would have the benefit that the notation is more compact and thus easier to follow. However, because an interface in UML cannot be instantiated, no multiplicity can be used in that case. When ports are used, it is possible to specify the number of instances of the ports related to an interface. In our case we need to be able to define multiplicity, for this reason ports need to be used. Another benefit of using ports is that it is possible to define alternative implementations of an interface. This has not been used, however.

Figure 5.4 shows an abstract component model for the PIM. The interfaces MMDevice, VDev and StreamEndPoint are closely related to each other and form a logical group. These interfaces are packed together in the abstract component EndPoint. This abstract component also implements the stream interface that is responsible for the actual streaming of the audio and the video. It is not specified how this abstract component is implemented, just that it implements the related interfaces.

The Controller abstract component just implements the StreamCtrl interface. This component can be located at the same node as one of the StreamParty components or at a separate node.

Figure 5.4: Abstract component model of the PIM

In the diagram the StreamInterface Stream is shown. This interface is introduced in this diagram and represents the actual audio and video streams. This diagram widens the scope a bit, because this interface has not been shown in any diagram before. This interface is also not defined in the Audio/Video Stream Specification. The reason why it is shown here is that together with the other interfaces it defines the component from a high abstraction level.
Since we want to distinguish different kinds of interfaces, we use different stereotypes for each interface type. We have used the same terminology as in the ODP-RM [6]; in Table 5.2 the different interfaces and their stereotype are listed.

Table 5.2: Types of interfaces

<table>
<thead>
<tr>
<th>Type of interface</th>
<th>Stereotype</th>
<th>Description in the ODP-RM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal interface</td>
<td>SignalInterface</td>
<td>An interface in which all the interactions are signals</td>
</tr>
<tr>
<td>Operation interface</td>
<td>OperationInterface</td>
<td>An interface in which all the interactions are operations</td>
</tr>
<tr>
<td>Stream interface</td>
<td>StreamInterface</td>
<td>An interface in which all the interactions are flows</td>
</tr>
</tbody>
</table>

5.2 Dynamic View

The actions as used in the CORBA PSM are nothing more than operation invocations. These actions have limited expressiveness. In order to model the actions more precise, we have used the Object Action Language.

5.2.1 Object Action Language

In the current version of UML action semantics have been incorporated. With the action semantics, actions as they are used in the dynamic view in the CORBA PSM can be specified with more detail. In Appendix B of the UML specification [28] examples in a number of action languages are given. We use the Object Action Language (OAL) [31] from Project Technology [32]. The choice for this language is more or less arbitrary, because all the other languages have comparable expressiveness and have a similar syntax.

This section gives some examples of usage of OAL, but it is not intended to give a complete specification. OAL is fairly similar to normal source code, but it is implementation-independent.

In OAL, control structures such as for each ... and if ... else ... can be used to control the execution flow. Local variables with types as integer, real, boolean and string can be defined. Furthermore, constructs used in the UML diagrams can be referenced and used in the code. The object diagram shown in Figure 5.5 is used for the examples of the usage of UML constructs in OAL.
For example, when the UML diagram includes the class \texttt{Employee} we can select an instance of that class by 'select any \( e \) from instances of \texttt{Employee}'. After this statement we have a variable \( e \) that refers to an instance of \texttt{Employee}, it is not specified to which instance if there are more instances. With 'select many \( es \) from instances of \texttt{Employee}' we select all of the instances of \texttt{Employee} in \( es \), which is in this case a set of references to instances of a class. With a 'for each \( e \) in \( es \) ... end for' loop the individual references can be used.

The attributes of a class can also be referenced. In our example, when we have a variable \( e \) with a reference to an instance of the class \texttt{Employee}, we can read the attribute \texttt{Name} with \( e.\texttt{Name} \). This can, for example, be used in an \texttt{if}-statement: \texttt{'if (e1.\texttt{Name} == "John Doe") \ldots end if'}. The attribute can also be set. This is done with, for example, \texttt{'e1.\texttt{Name} = "John Doe"'}.

In our example the class \texttt{Employee} is related to the class \texttt{Company} through the relationship 'works for'. When we have a reference \( e \) to a certain instance of \texttt{Employee} we can then get a reference to the related instance of \texttt{Company} with 'select any \( c \) related by \( e \to \texttt{Company}[\texttt{works for}]\)'.

Invocations of operations can be made by using the name of the operation. For example, when we have a variable \( c \) that is a reference to a \texttt{Company} Object, we can use '\texttt{c.sellGoods()}' to invoke the operation \texttt{sellGoods}.

For more information and for the specification of the language, Project Technology offers a manual [31] for download on their website [32].

### 5.2.2 Statechart Diagrams

We have four classes for which we need to create a statechart diagram: \texttt{StreamCtrl}, \texttt{MMDevice}, \texttt{VDev} and \texttt{StreamEndPoint}. Each statechart diagram is based on the corresponding statechart diagrams of the CORBA PSM. Because we have combined some classes in the PIM we also combined the statechart diagrams for these classes.
In the statechart diagram of the PSM various error states are used. In these states nothing happens, the object just returns to the previous state and from there everything continues. Most of these errors are related to QoS, multicasting or the Simple Flow Protocol and would have been deleted in any case, because the placeholders that are present in the CORBA PSM are ignored in the PIM. Instead of explicitly ignoring the operation invocations that cause the errors by the error states, we deleted the error states and implicitly ignore them in the PIM. This results in much cleaner diagrams that are easier to follow.

We can use the same states in the statechart diagram for StreamCtrl as we have used in the statechart for VSoDStreamCtrl in the CORBA PSM. The OCL constrains are also still valid.

The difference between the statechart diagrams of the CORBA PSM and the statechart diagrams in the PIM is the way in which actions that occur after a certain operation is called are modeled. In the CORBA PSM these actions are listed after the operation name and after the \(/-\)-symbol, in the statechart diagram of the PIM these actions are specified in OAL shown in a comment box connected to the operation name. In Figure 5.6 the new statechart diagram for the StreamCtrl object is depicted.

![Statechart diagram for StreamCtrl](image)

The five comment boxes with OAL code in Figure 5.6 are discussed below. The first comment box is connected to the operation bind_devs. In the CORBA PSM the line:

```
  m_a.create_A, m_b.create_B, bind
```

is used. In the PIM this line becomes the following OAL fragment:

```
s_a = m_a.create_A();
s_b = m_b.create_B();
bind();
```

The operation calls are transformed directly, the (); behind the operation name is purely a syntactical difference. A true difference is that in OAL the result of the operation invocation
is explicitly stored. The variables \( s_a \) and \( s_b \) which contain references to \( \text{MMDevice} \) objects are used in other operations.

The line:

\[
vdev_a.set_peer, vdev_b.set_peer, s_a.connect
\]

is transformed into:

\[
\text{select any } vdev_a \text{ related by } s_a \rightarrow \text{VDev};
\]

\[
\text{select any } vdev_b \text{ related by } s_b \rightarrow \text{VDev};
\]

\[
\text{relate } s_a \text{ to } s_b \text{ across } R4;
\]

\[
\text{relate } vdev_a \text{ to } vdev_b \text{ across } R5;
\]

\[
\text{relate } vdev_b \text{ to } vdev_a \text{ across } R5;
\]

\[
vdev_a.set_peer();
\]

\[
vdev_b.set_peer();
\]

\[
s_a.connect();
\]

The last three lines are essentially the same as the original line and only differ in a syntactical sense. The statements above these three lines first select the two \( \text{VDev} \) objects that are related to the \( \text{StreamEndPoint} \) objects, then relate the \( \text{StreamEndPoint} \) object to each other and finally relate the selected \( \text{VDev} \) objects to each other. The names of the relationships, \( R4 \) and \( R5 \), are defined in the class diagram depicted in Figure 5.3.

The creation of relationships is not modeled in the CORBA PSM. The information about the relationships is found in the code. This could have been prevented in case the CORBA PSM would also make use of OAL. This has not been done for practical reasons.

The other comment boxes with OAL statements are not interesting to discuss. In all cases two statements used in the CORBA PSM are directly transformed into a fragment of OAL code which consists of the same two statements.

The CORBA PSM has two different specializations of the \( \text{MMDevice} \) class, one for each party and those specializations implemented only \( \text{create}_A \) or \( \text{create}_B \), but not both. In the PIM the two classes are combined, so a new \( \text{MMDevice} \) object should support the dynamic aspects of both the classes in the CORBA PSM. Consequently, an \( \text{MMDevice} \) object should implement both the operations \( \text{create}_A \) and \( \text{create}_B \).

The statechart diagrams of \( \text{MMDevice}_A \) and \( \text{MMDevice}_B \) both have the states \( \text{Initialized} \), \( \text{VSoDStreamCtrl created} \), and \( \text{Ready} \). In the two diagrams the states \( \text{Initialized} \) and \( \text{VSoDStreamCtrl created} \) have the same semantics in each diagram. The state \( \text{Initialized} \) means that the object is just initialized and nothing has happened yet and the state \( \text{VSoDStreamCtrl created} \) means that there certainly exists a \( \text{VSoDStreamCtrl} \) object. The semantics of the state \( \text{Ready} \) differs for both diagrams. In \( \text{MMDevice}_A \) this state means that the object has created all the objects that are needed for party A. In \( \text{MMDevice}_B \) this state means that the object has created all the objects that are needed for party B.

Because the semantics of the state \( \text{Initialized} \) is the same in \( \text{MMDevice}_A \) and \( \text{MMDevice}_B \), in the new statechart diagram only one state is needed; this state is called \( \text{Initialized} \). The same holds for the state \( \text{VSoDStreamCtrl created} \). Figure 5.7 depicts the new statechart diagram.
The operation `bind` just creates a `StreamCtrl` object and calls the operation `bind_devs` on the newly created object. In the PSM the code for this was:

```plaintext
<<create>> sc, sc.bind_devs
```

and this has been transformed into:

```plaintext
create object instance sc of StreamCtrl;
sc.bind_devs();
return sc;
```

At the first line a `StreamCtrl` object is created. In OAL this object is explicitly typed, in the PIM the type of the `VSoDStreamCtrl` is only implicitly known. In OAL the newly created `StreamCtrl` object is explicitly returned to the caller of the operation.

The other operations, `create_A` and `create_B` are very similar to each other. For this reason we only discuss `create_A`. In the CORBA PSM the creation of the objects was captured in this one line:

```plaintext
<<create>> vdev_a, <<create>> sep_a
```

In the PIM not only the creation of the objects is modeled, but also the creation of the relationships between the newly created objects. In the last line of the following fragment the attribute `available_protocols` is set to the “RTP”. This was not specified in the CORBA PSM, because only operation invocations and creation of objects were modeled.

```plaintext
create object instance s_a of StreamEndPoint;
create object instance vdev_a of VDev;
relate self to vdev_a across R2;
relate vdev_a to s_a across R3;
s_a.available_protocols = "RTP";
```

The specification supports the binding of just a subset of the flows by using the argument `the_spec`. This argument, which is used in many operations, is an array that is either empty or contains the names of the flows that need to be bound. Because the implementation always binds all the flows and does not implement this feature this also has not been modeled in the CORBA PSM.
When the\_spec is empty and all Flows should be bound, all Flows connected to the MMDevice are selected and are related to the VDev. When the\_spec is not empty, for all of the Flows connected to MMDevice which are in the\_spec a connection is made to the VDev.

```c
if (rcvd_evt.the\_spec == "")
    -- when no the\_spec is given, create a vdev with
    -- all the capable flows
    select many cap\_flow\_set related by self->Flow;
    for each cap\_flow in cap\_flow\_set
        relate vdev\_a to cap\_flow across available;
    end for
else
    -- when the\_spec is given, create a vdev with only
    -- the flows in the\_spec
    select many cap\_flow\_set related by self->Flow;
    for each cap\_flow in cap\_flow\_set
        num_specs = 0;
        for each spec in rcvd_evt.the\_spec
            if (cap\_flow == spec)
                num_specs = 1;
            end if
        end for
        if (num_specs == 1)
            relate vdev\_a to cap\_flow across available;
        end if
    end for
end if
return s\_a;
```

The last part of the code is a bit awkward. In OAL no construct exists to test whether a value exists in a collection. In case such a construct was available, a line such as
```c
if (cap\_flow in rcvd_evt.the\_spec)
```
could have been used to select the Flows that exist in the\_spec. This would have prevented us from committing to a particular implementation of the search in a collection at this point in the trajectory.

For the classes VDev and StreamEndPoint also statechart diagrams are created. In these diagrams no additional concepts are used so they are not discussed here.
6 Web Services Specific Models

This chapter discusses the Web Services PSM, which corresponds to the third step in our development trajectory. This PSM is created in order to be able to implement the application on the Web Services platform. Our PIM from Chapter 5 has been transformed into this PSM. This chapter discusses the steps taken to create the PSM.

Similarly to the CORBA PSM, we create multiple PSMs for Web Services: an Interface PSM that models the Web Service interfaces and a Realization PSM that models the implementation choices made for the implementation of the interfaces. These PSMs are discussed in Section 6.1 and Section 6.2, respectively.

6.1 Interface PSM

The interface for a Web Service is defined in WSDL. In order to implement and deploy a Web Service, a WSDL interface should be created at some point in the trajectory. There are two alternatives for this. The first option is to create a UML model customized for Web Services first and generate the WSDL interface from that model. The second option is to generate the WSDL interface directly from the PIM. In Figure 6.1 both alternatives are depicted.

For the first approach we have to define a model that holds enough information to be able create the WSDL interface. The creation of this intermediate model allows us to model behavior using the OAL, as well as to use other complementary UML diagrams. This enables us to capture much more information than directly generating the WSDL interface. For this reason the first approach has been used. Therefore, we define a simple WSDL UML profile.
that is discussed in the next section. We present the mapping rules grouped according to constructs and patterns present in the PIM.

6.1.1 WSDL and a WSDL UML Profile

Web Services interfaces may be described in WSDL [39]. In a WSDL description, the following concepts are used:

- **Type**: a container for data type definitions using some type system;
- **Message**: an abstract, typed definition of the data being communicated;
- **Operation**: an abstract description of an action supported by the service;
- **Port Type**: an abstract set of operations supported by one or more endpoints;
- **Binding**: a concrete protocol and data format specification for a particular port type;
- **Port**: a single endpoint defined as a combination of a binding and a network address;
- **Service**: a collection of related endpoints.

It should be possible to map the information in the class diagram that is created in the WS PSM to those concepts in such a way that a complete interface can be constructed using WSDL. This does not mean that every concept in WSDL has to be graphically represented in the class diagram. For example, it is possible to specify that every Port Type uses the same kind of protocol and data format specification. In that case, the Binding does not have to be shown in the diagram. When the kind of protocol and data format used are known, it is possible to generate the binding parts in the WSDL file for each Port Type. This information can be specified in transformation parameters.

A Port is an instance of a Port Type. At this abstraction level we do not want to specify on which port and on which node an instance of a Port Type should be run. So, we do not model Ports.

An operation consists of a Message for the operation invocation which carries the arguments and possibly a Message for the return value. For this reason we do not model Messages explicitly, but the Messages in a WSDL file can be constructed from the arguments and the return values of the operations.

We conclude that Types, Operations, Port Types and Services can be graphically represented in the class diagram. Modeling these concepts is enough to be able to create an interface definition using WSDL. Table 6.1 shows how we have represented these concepts.

<table>
<thead>
<tr>
<th>WSDL concept</th>
<th>Graphical representation in UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Class with attributes. The stereotype WSDLType is used to denote that a class is a Web Services type.</td>
</tr>
<tr>
<td>Operation</td>
<td>Operation with specified parameters and return type. The stereotype WSDLOperation is used to denote that the operation is a Web Service operation.</td>
</tr>
<tr>
<td>Port Type</td>
<td>A class with stereotype WSDLPortType. The Operations contained in the Port Type are depicted as operations in the class.</td>
</tr>
<tr>
<td>Service</td>
<td>A class with stereotype WSDLServiceType. The Port Types indirectly associated to the Service are connected to the Service using an aggregation.</td>
</tr>
</tbody>
</table>

Not all classes can be used as Type. Only classes that can be created using the mapping rules from Java API from the XML-based Remote Procedure Call (JAX-RPC) specification [36] should be used. This does not mean that JAX-RPC must be used, it only defines the possible
classes that can be used. Basic types such as integers and strings are not defined. These types are mapped implicitly to the basic types defined in the XML Schema [43].

In WSDL a Service is a collection of Ports. Because we use only Port Types we cannot use the construct Service directly. However, because for each Port Type only one Port is created, we can use a virtual Service Type as a collection for Port Types.

The used stereotypes are not specified in any existing standard profile and they have been based on [9]. In Figure 6.2 the last three concepts can be seen. In this fragment one interface from the PIM is transformed into an incomplete Interface PSM.

![Figure 6.2: Graphical representation of one transformed RemoteObject](image)

6.1.2 Mapping RemoteObjects, operations and attributes

The class diagram in the static view of the PIM consists of RemoteObjects and some local classes. Each RemoteObject can be split into interfaces and implementing classes. For deriving the Web Services Interface PSM, we use the interfaces and the local classes only. The implementing classes are discussed during the construction of the Realization PSM. The interfaces from the RemoteObjects and operations can be transformed from the PIM to the PSM directly.

In the PIM the classes Flow and Format have attributes. The PIM does not specify how these attributes are used. In the Web Services PSM, setters and getters are defined for each attribute. These are operations that set or get the value of the attribute, respectively. Because we do not have to change the attributes we do not need the setters to be reachable remotely for these attributes. The decision whether the set and get operations need to be reachable remotely has to be made by hand. The outcome of that decision should be passed as a parameter of the transformation, so that during the transformation this information can be used and the appropriate set and get operations can be generated.

Each interface from a RemoteObject in the PIM is transformed into a Port Type. The operations of a RemoteObject are transformed into an Operation. The RemoteObjects MMDevice, VDev and StreamEndPoint in the PIM form a larger group. In the PIM they are also related to the same AbstractComponent in the abstract component diagram. At the Interface PSM deployable units are used. We have created a Service Type for each AbstractComponent from the PIM. Another approach was to create a Service Type for each interface in a
RemoteObject. However, our approach has the advantage that more coarse-grained services are created, which is good practice when using Web Services.

6.1.3 Mapping Identifiers

Web Services are just providers of operations. This means that it is not possible to use an Object Identifier on which operations can be called. To simulate this behavior each operation is given an additional parameter to denote the instance of the object connected to the Web Service. This parameter is called UID and is an integer. The application has to make sure that every instance is given a unique UID.

Sometimes it is necessary to pass a reference to an instance as an argument of an operation, or to give a reference to an instance back as a result of an operation invocation. In those cases it is not only necessary to use the UID, but also to use the Uniform Resource Locator (URL) of the Web Service. For this a type WS_ID has been created. This type consists of two attributes, UID and URL. Because this type can be used to specify a general Web Service instance and in many cases one wants to specify a specific Web Service instance, for each Service an empty sub-type is created. These sub-types are used for type checking. This is optional and it would also be possible to use only the general WS_ID type. Also, it would have been possible to use a completely other scheme, such as using a simple string that encodes the URL of the Web Service together with the identifier.

6.1.4 Mapping the State

An important property of an object is its state. A state of an object can be defined as the active state in the statechart diagram defined for the object together with the values of the variables of the object.

The implementation of an object can either be stateless or stateful. When the implementation is stateful the objects keep their state themselves. When the implementation of an object is stateless the state of the object is kept somewhere else, and needs to be transferred to the object when it is used. Both approaches have their advantages.

Let us consider the example of an application with a server object and several clients that access this server object. In the situation that the server is implemented as a statefull object, objects that have a state depending on the client, are used by each client. Because the server has to hold information for each active client in the stateful situation this approach is not very scalable. In cases when the number of clients is limited this is not a large problem. In the stateless situation the clients themselves hold the state information, this means that the server can be distributed over more nodes more easily. The scalability is thus much higher. This is the reason why this approach is considered best practice when using web services. In general, one does not know how many clients one must serve, and this number can become extremely high in case of a web service made available freely on the Internet. Scalability is in those situations extremely important.

In the stateless situation the client is responsible for keeping the state. In many cases, the developer has little or no control over the implementation of the clients. For example, the server developer may not be able to make sure that the state is not forged. When the state is forged by the client it is possible that the new state is impossible. This can, and should, be checked by the server. However, in case the forged state is just different from the original state, but is still valid, the server has no means to check this.

Another drawback of a stateless server is that the state has to be transferred with every operation invocation. This creates a large overhead in case the state is complex.
In our case we know that the number of active objects is small, scalability is thus not an issue. So, using stateless objects is not necessary and has no advantages over using stateful objects. For our system, it is more logical to let the object keep their state. This means that we do not need to add an argument in each operation that represents the state.

6.1.5 Resulting Class Diagram

In Figure 6.3 the complete class diagram is shown.

![Class Diagram](image)

**Figure 6.3: Class diagram of the Web Services PSM**

6.1.6 Dynamic View

For each PortType defined in Figure 6.3 we define a statechart diagram. In the Interface PSM of the PIM also statechart diagrams have been created, so each statechart diagram of a PortType is based on a statechart diagram in the PIM.

In Figure 6.4 the statechart for StreamCtrl is shown. This diagram is very similar to the model shown in Figure 5.6. The only difference is that the actions include arguments.
s_a = MMDevice.create_A("");  
s_b = MMDevice.create_B("");  
bind(s_a, s_b);  

context StreamCtrl::bind_devs  
pre: self.mmdevice.vdev.streamendpoint->size() = 0  
pre: self.mmdevice.vdev->size() = 0  
post: self.mmdevice.vdev.streamendpoint->size() = 2  
post: self.mmdevice.vdev->size() = 2  

select any vdev_a related by s_a -> VDev[R3];  
select any vdev_b related by s_b -> VDev[R3];  
relate s_a to vdev_a across R4;  
relate s_b to vdev_a across R4;  
relate vdev_a to vdev_b across R5;  
relate vdev_b to vdev_a across R5;  
VDev.set_peer(vdev_aUID, vdev_b);  
VDev.set_peer(vdev_bUID, vdev_a);  
StreamEndPoint.connect(s_aUID);  
StreamEndPoint.stop(s_aUID);  
StreamEndPoint.stop(s_bUID);  
StreamEndPoint.start(s_aUID);  
StreamEndPoint.start(s_bUID);  
StreamEndPoint.disconnect(s_aUID);  
StreamEndPoint.disconnect(s_bUID);  

Figure 6.4: Statechart diagram for StreamCtrl

In the static view we have defined the arguments for the operations, so now we can make use of those arguments and include the parameters at an operation invocation.

6.2 Realization PSM

We use a similar approach with the Realization PSM of the Web Services PSM as the class diagram of the Realization PSM of the CORBA PSM. A fragment is created that models only one object in detail. After that, this template can be used to create a complete class diagram in which instances of the fragments are collapsed into a more compact representation. Since the complete diagram is not very interesting it is not shown here. As the dynamic view of the system does not change with respect to the Interface PSM, we do not model it.

We have used JBuilder 7 with the Web Service Kit for Java [3] to create the template. This IDE provides the automatic creation of Java stubs from a WSDL interface file or from a normal Java class that has to be converted into a Web Service. This last option has been used to create all the stubs. This is because after the creation of the Interface PSM no WSDL interface is available. So, either the WSDL interface or the Java classes have to be written by hand.

For the creation of the template, first a class as depicted in Figure 6.5 was created using plain Java.

```
<interface>

<operation>()
<operation>()
```

Figure 6.5: The Java class used to create the stubs
This class was then transformed into a Web Service by performing the ‘Export Class as a Web Service’ action. This results in the classes and interfaces shown in Figure 6.6.

Figure 6.6: One Service and one Port Type implemented using JBuilder
7 Evaluation

This chapter evaluates some important points of the trajectory that has been followed. For these points design decisions are discussed.

We start with the differences found between PIMs and PSMs in Section 7.1. After that, we discuss the usage of UML and OAL in Sections 7.2 and 7.3, respectively. Finally, in Sections 7.5, 7.6 and 7.6 we discuss the usage of components, EDOC and transformations.

7.1 Differences between PIMs and PSMs

We have found two differences between a PSM and a PIM in our project. These are:

- the degree of platform independence in the model: to what degree are platform independent constructs used in the model. For example, in the PIM attributes are used to represent properties that are used in the CORBA PSM;
- the abstraction level of the model: to what degree information is omitted to make the model more clear. For example, at the PIM we have not considered different stream parties anymore.

Furthermore, we have found another difference between a PSM and a PIM. However, this is not a difference in general, but a difference that was specific for our project.

- the scope of the model: which parts of the complete system are modeled, and which are omitted. In theory, this should remain the same. However, in case that a system is ported and maintained, parts of a system may become obsolete or new parts need to be defined. In our project, the PIM supports more types of streams than the CORBA PSM.

7.2 The usage of UML

UML has been used to model all of the models in all of the PIMs and PSMs. In UML various different kinds of diagrams are defined. We have used only class diagrams for the static view and sequence diagrams and statechart diagrams for the dynamic view. The class diagrams and statechart diagrams together have been found sufficient to model the complete system, both at the PSM and at the PIM level.

We have made use of profiles in the various models. Unfortunately, only one predefined profile, the UML profile for CORBA, could be used. We had to define the other profiles ourselves. The class diagram that was created using the mappings from IDL defined in the UML profile for CORBA did not show relations between the interfaces created. For this, we have created another class diagram that extends the created diagram to show these relations. For the PIM, we have defined a profile that includes a class diagram similar to the class diagram in the CORBA PSM and an abstract component diagram. For the Web Services
PSM, we have defined a profile that includes a class diagram representing the interfaces defined in WSDL.

### 7.3 The usage of OAL

For defining actions in the PIM and in the Web Services PSM we have used OAL. The syntax of OAL does not have a higher abstraction level than the syntax for defining actions used in the CORBA PSM. The main difference is that the OAL is more expressive. It is for example possible to create links between objects.

The reason why OAL has not been used in the CORBA PSM is purely pragmatic. OAL has been identified as a useful modeling language by us after we already made the CORBA PSM. In a new project OAL should be used from the start.

One of the problems with OAL (and other action languages) mentioned in the literature is the low abstraction level [14]. At the PSM it is more or less a representation of the programming code. Although this critic is valid, in our experience OAL is useful for implementation oriented specification of behavior of objects.

In our trajectory the PIM with a low abstraction level was enough to fulfill our needs, because the PIM was used to model the Web Services PSM. In case we needed a PIM with a higher abstraction level, we could use the PIM that uses OAL to create a PIM with a higher abstraction level. For example, by creating a PIM in which the actions are replaced by pre- and post-conditions using OCL. This is schematically depicted in Figure 7.1.

![Figure 7.1: Abstraction level of the PIM](image)

A drawback of the usage of OAL is that it does not support parallelism. If this was the case, we would be able to avoid specifying some unnecessary ordering restrictions between actions.

### 7.4 The usage of Components

Nowadays, components are the coarse grained building blocks of applications. In case an application is built as a composition of components, those components can possibly be reused by other applications. Furthermore, it is sometimes possible to buy prefabricated components
from others, the so-called COTS (Commercial Off-The-Shelf) Components, and integrate them into the applications, thus saving valuable construction time.

Since CORBA version 2.x does not support components, the concept of components is not used in the CORBA PSM. We have chosen to introduce components in the PIM. Another option was not to use components in the PIM, but to include a more extensive mapping in the transformation from the constructs in the PIM to the Service Type in the Web Services PSM. Now, the mapping can be almost one-to-one. Another advantage of using components at the PIM is that more information is captured and the PIM can also be used for more things. For example, when the PIM is used for the creation of another PSM for a middleware technology that supports components the transformation can be much simpler than when no components are defined in the PIM.

7.5 The usage of EDOC

We have used only a standardized profile for the CORBA PSM. For the PIM and the Web Services PSM we have defined our own profiles. For the Web Services PSM there was no profile available and thus we did not have a choice. For the PIM, another approach would have been to use the UML Profile for EDOC [27].

An advantage of EDOC is that the specification already exists and that it can be used to model a system from a high abstraction level. A drawback is that the EDOC standard is several hundreds pages long and the developer thus has to learn to use yet another specification. Furthermore, the EDOC standard is not widely supported in tools.

EDOC also does not have support for streams. In our abstract component diagram in the PIM this is included, using concepts from the ODP-RM.

The learning curve for using MDA should be as small as possible, the longer it takes, the larger the chance that MDA is not used at all. The UML Profile for EDOC introduces entirely different semantics to the UML constructs. The profiles used in the PIM and in the Web Services PSM are as light as possible, allowing a new developer to start creating diagrams as soon as possible.

7.6 The usage of Transformations

In Section 2.4 four different transformation approaches are discussed:
1. by hand, by replacing constructs by others, without any systematic technique;
2. by hand, using established patterns to convert from the PIM to a particular PSM;
3. by a tool, implementing the established patterns, which produces a skeleton PSM, that is later completed by hand;
4. by a tool that produces the entire PSM.

The first approach has the least steep learning curve when you do not have any background to build on and you have to invent everything yourself. No knowledge of other transformations is required. However, a drawback of this method is that no knowledge is used and no knowledge is captured. When the MDA approach is used more than a few times, this method is less efficient than the others.

The second approach can be seen as an evolution of the first one. When MDA is used a number of times, in practice the transformation steps are known implicitly. So, in reality the
knowledge is captured. When this knowledge is formalized and patterns are created, these patterns can be used in future transformations. This limits the creativity of the designer, but increases the productivity.

In the third approach the patterns are made explicit in such a way that a tool can use them to generate most of the PSM. This implies that the mapping rules have to be made explicit in a certain format so that a tool can use them.

In this approach only a skeleton of the PSM is automatically constructed. This has the benefit that only for the most commonly used patterns in the PIM a precise mapping has to be defined. In practice the easy parts are mapped, while the complicated parts are ignored. So, with this approach the boring repetitive work is done automatically, while the interesting, creative work is left to the developer.

The fourth approach limits the time needed for performing the transformation by automating everything. This makes the MDA approach available in situations where things change quickly, a situation seen in many software creation projects nowadays.

A complete automated transformation implies that all the information needed to make the transformation is explicitly available. This makes the transformation less flexible and it is not always feasible. A consequence of this approach is that the PIM and the mapping contain much detailed information. Instead of introducing the complexity during the programming the complexity is already added during the creation of the PIM. This also means that the creation of the PIM looks more as programming than with the other three approaches. A drawback of this all is that one of the important features of a PIM, the usage of the PIM as a high-level overview, is lost.

In our project we did not use any tools for the transformations. We followed the first approach, in order to enable the second and the third. Before the second and the third approaches are practiced it is important to master the knowledge about the models and their relations that is needed to perform the transformations.
8 Conclusions

This chapter presents the conclusions that can be drawn from this project. Furthermore, it identifies points where further investigation is necessary.

This chapter is further structured as follows: in Section 8.1 we present our contributions, in Section 8.2 we give the conclusions that can be drawn from this project and in Section 8.3 we present possible future work.

8.1 Contributions

The main contributions of this project are:
- the identification of information that should be present in PIMs and PSMs;
- improving the understanding of the relationship between the different kind of models, both in a top-down and in a bottom-up MDA trajectory;
- the definition of the representation of PIMs and PSMs using UML, both of the static view and of the behavior.

8.2 General Conclusions

In the development of this thesis, we have used a case study that showed the re-engineering of an application that uses the Audio/Video Stream Specification on the CORBA platform to the Web Services platform. In this project we have shown that it is possible to create a model that is independent of the middleware technology used by the application, from a model specific for the middleware technology. Furthermore, we have shown that we could use this PIM to create a model for another middleware platform.

For this, we have first created a Platform Specific Model that models the existing application. An important characteristic of the CORBA PSM is that it consists of two complementary parts that are used in the rest of the trajectory: an Interface PSM that defines the static aspects of the interfaces defined in IDL, and an Object PSM that defines the static and dynamic aspects of the objects that implement the interfaces. First, the CORBA interface is modeled using a class diagram. After that, a model has been created to represent the CORBA Objects using class diagrams, with stereotypes in order to distinguish CORBA Objects from implementing classes, and statechart diagrams, for defining the complete behavior of the CORBA Objects. To create this last PSM we needed more information than is available in the interface definition defined in IDL. This information was obtained from the Audio/Video Stream Specification and from the existing implementation of the application.

After creating the CORBA PSM we have created the Platform Independent Model. The PIM models the system from a platform independent perspective. This model can be used as a starting point for the top-down part of the trajectory. Apart from the class diagram and
statecharts that are used for the same reasons as in the CORBA PSM, we have introduced an abstract component diagram to group related objects together in this model. A benefit of this approach is that the transformation to the Web Services PSM and to potential platforms that have a direct support for components can be easier.

In the PIM we have used OAL for specifying actions. This enables the standalone usage of the PIM in the second part of the trajectory. The PIM alone contains enough information to create the Web Services PSM. A drawback of using OAL in the PIM is that the action definitions in the PIM do not have a higher abstraction level than the action definitions in the PSM. In our trajectory this is not a large problem, because the primary reason the PIM is created is to be able to port a system from one middleware platform to another. In case we needed a model at a higher abstraction level, the PIM using OAL would just be the first step. After that, a model should be created that defines the actions in a more abstract way, for example, by using pre- and postconditions which only define the state before an operation invocation and the state after an operation invocation, without defining how the end-state should be achieved.

The Web Services PSM defines the system at a platform specific level and can be used for the implementation of the application using Web Services. The Web Services PSM consists of a static view and a dynamic view, just as the CORBA PSM and the PIM. In the Web Services PSM the class diagram created represents the WSDL interface. In order to do this, we had to make decisions about how to transform interfaces, operations, attributes and how to model the identifiers and the state of objects. The interfaces, operations and attributes could be transformed directly from the PIM. For the identifiers, we have created a type that makes it possible to identify a certain instance of an object uniquely. We have used stateful objects, this way the objects were more similar to the objects in the CORBA PSM and in the PIM. For that reason the transformation between the PSM and the PIM is simpler than when stateless objects were used.

One of the sub-goals in this project was to port an application that used the OMG Audio/Video Stream Specification from CORBA to Web Services. The approach followed was certainly not the most efficient one on the short term. It would have been more efficient to port the original application directly to the Web Services platform. However, such porting projects can be done much quicker with the knowledge gained during this project and with tools to assist the developer.

Furthermore, the separation of the PIM and the PSM helped us to identify limitations of the models because of available middleware constructs. For example, in CORBA 2.x (abstract) components do not exist. With MDA, we could introduce this construct at the PIM layer, so that abstract components could be used in the transformation to the Web Services platform. Also, because no stream support is available in CORBA, this has not been modeled in the CORBA PSM. Finally, using MDA we have found platform-specific patterns, such as the use of the Property Service in CORBA and we identified alternatives for it in the Web Services realization.

8.3 Future Work

Several improvements and extensions to this project can be made. In this section a number of these are presented.

The next step after the creation of the Web Services PSM in the trajectory is to create an implementation using Java and Web Services. The creation of the implementation should show that it is really possible to port a system from one middleware platform to another using MDA. A comparison of the implementation obtained by following the MDA trajectory with
an implementation obtained by porting the application directly to a new platform may be useful to investigate to what degree MDA is useful for such a trajectory.

In the near future the standardization of UML version 2.0 will be finished. Most of the diagrams created during the project can remain the same, but some changes have to be made to make all the diagrams compliant with UML version 2.0. Furthermore, in UML 2.0 the concept of abstract component is supported. So, it may be possible to replace the abstract component diagram discussed in Section 5.1.3 by a native UML 2.0 component diagram, or to create a profile in which such a diagram is used.

Before tools can be used to automate the trajectory used, it is important to understand how the models should be defined and how the models are related to each other. After this is done, tool support should follow this activity. In particular, tool support should be investigated for the creation of models using profiles and the automation of transformations.

The action language used in the PIM and in the Web Service PSM does not support parallelism. Because of this, the action language could not be used to model the PIM at a higher abstraction level. Although parallelism can now be modeled using concurrent sub-states, this is not practical. The usage of parallelism in OAL and the usage of other languages as replacement for OAL should be investigated.
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMIDST</td>
<td>Application of Middleware in Services for Telematics</td>
</tr>
<tr>
<td>CCA</td>
<td>Component Collaboration Architecture</td>
</tr>
<tr>
<td>CIM</td>
<td>Computational Independent Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CWM</td>
<td>Common Warehouse Metamodel</td>
</tr>
<tr>
<td>EDOC</td>
<td>Enterprise Distributed Object Computing</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>JAX-RPC</td>
<td>Java API for XML-based Remote Procedure Call</td>
</tr>
<tr>
<td>JMF</td>
<td>Java Media Framework</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OAL</td>
<td>Object Action Language</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>ODP-RM</td>
<td>Open Distributed Processing Reference Model</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-Time Transport Protocol</td>
</tr>
<tr>
<td>SFP</td>
<td>Simple Flow Protocol</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VSoD</td>
<td>Video Streaming on Demand</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Meta-Data Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformations</td>
</tr>
</tbody>
</table>
References

[8] L. Ferreira Pires, Middleware for Distributed Objects, Lecture 1: Introduction, University of Twente, Enschede, The Netherlands, August 2002


Appendix A: Additional Diagrams

During the project many diagrams were created. Because we have shown only the most interesting diagrams in the report, many of the diagrams created are not shown yet. In this appendix these diagrams are shown.

CORBA PSM Diagrams

Figure A.1: Statechart diagram for MMDevice_A in the CORBA PSM

Figure A.2: Statechart diagram for MoviePlayerVDev in the CORBA PSM
Figure A.3: Statechart diagram for MovieReaderVDev in the CORBA PSM

Figure A.4: Statechart diagram for ReceiverStreamEndPoint in the CORBA PSM
PIM Diagrams

Figure A.5: Statechart diagram for TransmitterStreamEndPoint in the CORBA PSM

```
// bind IP address and portnumbers
// stand by for receiving
// bind IP address and portnumbers
// stand by for transmitting
select any rel_sep related by self -> StreamEndPoint[R4];
rel_sep.request_connection();
```

Figure A.6: Statechart diagram for StreamEndPoint in the PIM

```
select any rel_sep related by self -> StreamEndPoint[R4];
rel_sep.start();
rel_sep.stop();
```

```
select any rel_sep related by self -> StreamEndPoint[R4];
rel_sep.stop();
```
-- for VDev/SEP with two flows (audio1 and video1), but a bit generalized
-- so it will work in the cases when the flows are exactly the same
-- on both sides and all the flows have to be connected.

select any vdev_related by self->VDev[R5];
select any own_sep related by self->StreamEndPoint[R3];
select any other_sep related by vdev_related->StreamEndPoint[R3];
select many own_flow_set related by self->Flow[available];
select many other_flow_set related by vdev_related->Flow[available];

for each own_flow in own_flow_set
found = false;
for each other_flow in other_flow_set
  if (own_flow.name == other_flow.name)
    found = true;
    if (own_flow.dir == other_flow.dir)
      return false;
    end if
  end if
if (found == false)
  return false;
end if
end for

select many own_cur_flow_set related by own_sep->Flow[current];
select many other_cur_flow_set related by other_sep->Flow[current];

for each own_cur_flow in own_cur_flow_set
  format_ok = false;
  select any current_format related by own_cur_flow->Format[current]
  for each other_cur_flow in other_cur_flow_set
    if (own_cur_flow.name == other_cur_flow.name)
      select any other_format related by other_cur_flow->Format[current]
      if (other_format == current_format)
        format_ok = true;
      end if
    end if
  end for
if (format_ok == false)
  select many own_pos_format_set related by own_cur_flow->Format[available]
  for each own_pos_format in own_pos_format_set
    for each other_cur_flow in other_cur_flow_set
      if (own_cur_flow.name == other_cur_flow.name)
        select many other_pos_format_set related by other_flow->Format[available]
        for each other_pos_format in other_pos_format_set
          if (other_pos_format == own_pos_format)
            relate own_cur_flow to own_pos_format across current;
            relate other_cur_flow to other_pos_format across current;
            format_ok = true;
          end if
        end for
      end if
    end for
  end if
if (format_ok == false)
  return false;
end if
end for

---

Figure A.7: Statechart diagram for VDev in the PIM
Web Services Diagrams

Figure A.8: Statechart diagram for MMDevice in the Web Services PSM

Figure A.9: Statechart diagram for StreamEndPoint in the Web Services PSM
select any vdev_related by self->VDev[R5];
select any own_sep related by self->StreamEndPoint[R3];
select any other_sep related by vdev_related->StreamEndPoint[R3];
select many own_flow_set related by self->Flow[available];
select many other_flow_set related by vdev_related->Flow[available];

for each own_flow in own_flow_set
    found = false;
    for each other_flow in other_flow_set
        if (own_flow.name == other_flow.name)
            found = true;
        end if
    end for
    if (found == false)
        return false;
    end if
end for

select many own_cur_flow_set related by own_sep->Flow[current];
select many other_cur_flow_set related by other_sep->Flow[current];

for each own_cur_flow in own_cur_flow_set
    format_ok = false;
    select any current_format related by own_cur_flow->Format[current]
    for each other_cur_flow in other_cur_flow_set
        if (own_cur_flow.name == other_cur_flow.name)
            select any other_format related by other_cur_flow->Format[current]
            if (other_format == current_format)
                format_ok = true;
            end if
        end if
    end for
    if (format_ok == false)
        select many own_pos_format_set related by own_cur_flow->Format[available]
        for each own_pos_format in own_pos_format_set
            for each other_cur_flow in other_cur_flow_set
                if (own_cur_flow.name == other_cur_flow.name)
                    select many other_pos_format_set related by other_flow->Format[available]
                    for each other_pos_format in other_pos_format_set
                        relate own_cur_flow to own_pos_format across current;
                        relate other_cur_flow to other_pos_format across current;
                        format_ok = true;
                    end if
                end if
            end for
        end for
    end if
if (format_ok == false)
    return false;
end if

Figure A.10: Statechart diagram for VDev in the Web Services PSM