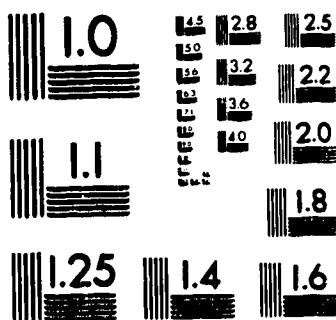


1

PM-1 3½" x 4" PHOTOGRAPHIC MICROCOPY TARGET  
NBS 1010a ANSI/ISO #2 EQUIVALENT



PRECISION<sup>SM</sup> RESOLUTION TARGETS



National Library  
of Canada

Acquisitions and  
Bibliographic Services Branch

395 Wellington Street  
Ottawa, Ontario  
K1A 0N4

Bibliothèque nationale  
du Canada

Direction des acquisitions et  
des services bibliographiques

395, rue Wellington  
Ottawa (Ontario)  
K1A 0N4

*Your file    Votre référence*

*Our file    Notre référence*

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

# **Object-Oriented Modeling in Metaview**

BY

**Yong Zhuang**



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

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta  
Fall 1994



National Library  
of Canada

Acquisitions and  
Bibliographic Services Branch

395 Wellington Street  
Ottawa, Ontario  
K1A 0N4

Bibliothèque nationale  
du Canada

Direction des acquisitions et  
des services bibliographiques

395, rue Wellington  
Ottawa (Ontario)  
K1A 0N4

*Votre* *lib* - *Votre référence*

*Votre* *lib* - *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-315-95142-7

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Yong Zhuang**

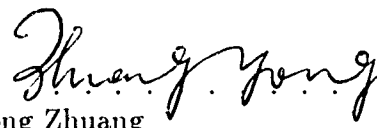
TITLE OF THESIS: **Object-Oriented Modeling in Metaview**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1994**

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

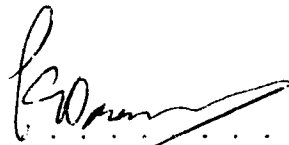
(Signed) . . .  . . . . .  
Yong Zhuang  
Kunming Medical College  
Kunming, Yunnan  
P.R.China

Date: *June 27th, 94*

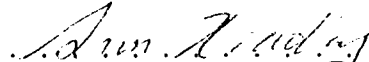
UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

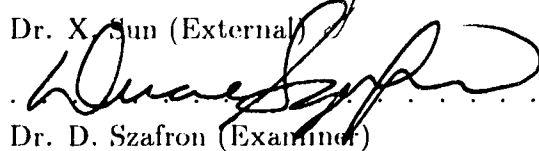
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Object-Oriented Modeling in Metaview** submitted by Yong Zhuang in partial fulfillment of the requirements for the degree of Master of Science.



.....  
Dr. P. Sorenson (Supervisor)



.....  
Dr. X. Sun (External)



.....  
Dr. D. Szafron (Examiner)

Dr. F.J. Pelletier (Chair)

Date: *June 27th., 94*

*DEDICATED*

*to my parents, brothers,  
and all who love me and beloved*

## **Abstract**

Computer-Aided Software Engineering (CASE) and Object-Oriented Development (OOD) are used to help efficiently construct reliable and maintainable software systems. This thesis focuses on creating object-oriented support environments in Metaview, a metasystem that can generate automatically a variety of environments to support software engineering activities, such as requirement analysis and design.

Although some structured environments have been defined in Metaview, no object-oriented methods have been modeled. An investigation of object-oriented modeling in Metaview achieves two purposes. Firstly, Metaview's capability to define object-oriented CASE environments is studied and critiqued with the objective of determining those modeling aspects that can be improved. The second major contribution of this thesis is the development of an enhanced object-oriented development method that is an amalgamation of several representative object-oriented methods. On the basis of the review and critique of popular O-O methods, an enhanced method, called the OOM (Object-Oriented Modeling) approach, is proposed, defined and prototyped in Metaview.



## Acknowledgements

I am very grateful to my supervisor, Professor Paul Sorenson, for his invaluable guidance, constant encouragement and patience throughout the research period. I feel lucky that he supervised my thesis. Thank also to the other members of my committee, Dr. Duane Szafron and Dr. Xiaolin Sun, for their valuable comments.

Thanks to all the Software Engineering Lab members. The friendly group atmosphere makes the thesis research enjoyable. Piotr Findeisen was always willing to provide assistance with unfamiliar parts of the Metaview System. I am grateful to Narendra Ravi, Kent McPhee, and Lettice Tse for useful discussions and various help.

I wish to express my affectionate gratitude to my dad Yunguang Zhuang and mom Qiongyao Hu, my brothers Wei and Min, my aunt Nianbei and uncle Guolian, for their constant love and support, for never doubting in me, always being proud of me, and never letting me forget it. Without their teaching, encouragement and moral support, I would not gone this far. Their love is one of the most important parts of myself. Deep appreciation also goes to other relatives and close friends who encourage me to make great dreams come to true, though I cannot list your names one by one here.

In no particular order, I thank my friends who made my life in Edmonton enjoyable and memorable: Xiaolin Cheng, Donna Kuhn, Kaladhar Voruganti, Yao Li, Leung Chu Kwong, and Rohrick family. Special thanks to Qunjie Wang for helping me in countless ways, for the understanding, encouragement and affection.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	CASE Environments and Metaview . . . . .	2
1.2	Basic Concepts of the O-O Method . . . . .	3
1.3	Outline of the Thesis . . . . .	4
<b>2</b>	<b>Metaview, the Generator of Multiple Environments</b>	<b>5</b>
2.1	Fundamental Concepts . . . . .	5
2.2	Architecture of Metaview . . . . .	7
2.3	Metaview Modeling Mechanism . . . . .	9
2.3.1	Method Modeling . . . . .	9
2.3.2	Environment Definition Guidelines . . . . .	11
<b>3</b>	<b>Modeling Rumbaugh's OMT in Metaview</b>	<b>12</b>
3.1	Purpose Of Modeling OMT . . . . .	14
3.2	OMT's OOA . . . . .	15
3.2.1	The Object Model . . . . .	16
3.2.2	The Dynamic Model . . . . .	25
3.2.3	The Functional Model . . . . .	28
3.2.4	OOA Models Integrity . . . . .	30

3.3	Modeling OMT's OOA . . . . .	31
3.4	Defining and Modeling the OOD . . . . .	45
3.5	Results and Discussion of OMT Modeling . . . . .	47
<b>4</b>	<b>O-O Methods Review</b>	<b>50</b>
4.1	Rumbaugh's OMT and OMTool . . . . .	51
4.2	Booch's Method and Rational Rose . . . . .	51
4.2.1	Models in Booch's Method . . . . .	52
4.2.2	Summary and Critique . . . . .	54
4.3	Fusion Method . . . . .	55
4.3.1	OOA . . . . .	55
4.3.2	OOD . . . . .	58
4.3.3	Summary and Critique . . . . .	62
4.4	Shlaer-Mellor's Method and Teamwork . . . . .	64
4.4.1	OOA . . . . .	64
4.4.2	OOD . . . . .	67
4.4.3	Summary and Critique . . . . .	69
4.5	Conclusion . . . . .	69
<b>5</b>	<b>Enhanced Method—OOM and Its Modeling</b>	<b>70</b>
5.1	OOM Method . . . . .	71
5.1.1	OOA Models . . . . .	72
5.1.2	OOD Models . . . . .	75
5.1.3	Implementation Phase . . . . .	77
5.1.4	Transitions Between OOM Models . . . . .	77
5.1.5	The OOM's Flexibility . . . . .	79

5.2	OOA Notation and Its Modeling . . . . .	79
5.2.1	Subsystem-Level OOA Models . . . . .	79
5.2.2	System-Level OOA Models . . . . .	81
5.3	OOD Notation and Its Modeling . . . . .	84
5.4	Defining Transformations Across Models . . . . .	90
5.5	The Experience of Defining OOM Using Metaview . . . . .	94
5.6	Summary . . . . .	96
<b>6</b>	<b>Conclusions and Future Research</b>	<b>100</b>
6.1	Conclusions and Contributions . . . . .	101
6.2	Suggestions of Future Research . . . . .	104
	<b>Bibliography</b>	<b>106</b>
<b>A</b>	<b>Part of the Definition for OOM Support Environment</b>	<b>110</b>
A.1	Conceptual Definition for Three OOA Models . . . . .	111
A.2	Process Model's Graphical Definition . . . . .	126
A.3	Some Across-Model Constraints . . . . .	129

## List of Figures

1	Aggregate in EARA model . . . . .	6
2	Current prototype configuration of Metaview . . . . .	7
3	OMT method . . . . .	13
4	The notation of class and object instance . . . . .	17
5	The notation of multiplicity and ordering . . . . .	17
6	The notation of qualified association . . . . .	18
7	The notation of ternary . . . . .	19
8	The notation of aggregation . . . . .	19
9	The notation of link attribute . . . . .	20
10	Derived attribute and operation . . . . .	21
11	Propagation of operations . . . . .	21
12	The notation of inheritance/generalization . . . . .	22
13	The notation of constraints . . . . .	24
14	State transition and sending events . . . . .	27
15	Synchronization of concurrent activities . . . . .	28
16	The basic notation of Data Flow Diagram . . . . .	30
17	Entities in Object Model . . . . .	31
18	Some entity notations in Object Model . . . . .	33

19	Relationship notation in Object Model . . . . .	34
20	Notation of Dynamic Model . . . . .	40
21	Notations of Functional Model . . . . .	43
22	Decomposing system . . . . .	46
23	Booch's Method . . . . .	52
24	An example of Module Diagram . . . . .	54
25	Fusion's OOA . . . . .	56
26	Object instance and collection . . . . .	59
27	Message sequencing . . . . .	59
28	Visibility . . . . .	60
29	Server binding and reference mutability . . . . .	61
30	The Fusion Method . . . . .	63
31	Shlaer-Mellor's OOA . . . . .	65
32	Shlaer-Mellor's OOD architecture . . . . .	67
33	Process of an application class's state transition . . . . .	68
34	OOM architecture . . . . .	71
35	Domain Analysis Model . . . . .	81
36	Object Communication Model . . . . .	84
37	Object Visibility Model . . . . .	86
38	Aggregation to create strong link . . . . .	91

**List of Tables**

4.1	Relationship Between Fusion and Other O-O Methods . . . . .	55
5.2	The measurement of OMT modeling . . . . .	98
5.3	The measurement of OOM modeling . . . . .	98

# Chapter 1

## Introduction

The dramatically increasing complexity of many software systems has led to the recent “crisis” of software development. *Computer-Aided Software Engineering* (CASE) has evolved to help solve this problem by providing automated assistance for development teams to manage this complexity. Because it is costly to produce efficient and reliable support environments, meta system facilities, such as *Metaview* [McA88, DeD91, Sor91, Sor88, Fin93e], *MetaPlcx* [CN89], *MetaEdit* [Smo91] and *Socrates* [Ver91], are being developed to generate automatically a variety of software specification environments to support major parts of CASE activities, such as requirement analysis and design. A common characteristic of metasystem is three-level architecture: meta level, environment definition level, and the application (or user) level. This architecture is described in greater detail in chapter 2 of the thesis. Each metasystem has a underlying model as a framework to develop software specification environments that are defined formally and unambiguous. This makes modeling time and cost of developing a specification environment significantly reduced. Moreover, it is easy to improve previously developed specification environments and to compare alternative environments in a metasystem [ST93]. At present, to our knowledge, there is no commercial products that use the metasystem approach.

The traditional structured methods of software development suffer from problems [KM90] that include little or no iteration within development phases, no emphasis on reusability and extendibility issues, and no unifying models and notation to integrate well the development phases. In response, the object-oriented (O-O) methods are proposed [Rum94, HS91]. The O-O approach is fast becoming the standard for



software system development in the industry.

It is significant to define object-oriented (O-O) environments in Metaview because no O-O environment has been implemented using Metaview. Although a number of O-O methods have been proposed, no standard has emerged and each existing O-O method suffers from some disadvantages. Therefore, it is also valuable to propose an enhanced O-O method that combines many of the best features from the existing methods. Because of OOM's incorporation of different O-O methods, prototyping the OOM method using Metaview can provide a broader test for Metaview's O-O modeling capability. The major thrusts of this thesis are to:

- explore the mechanism of defining object-oriented environments in Metaview. The challenge of integrating O-O development methods in a common environment can stress test the modeling capabilities of Metaview;
- propose an enhanced O-O method and prototype it in Metaview.

Section 1.1 provides a brief background of Metaview. Section 1.2 introduces the O-O development methodology and its basic concepts. The chapter concludes with an overview of the remaining chapters of the thesis.

## 1.1 CASE Environments and Metaview

Several key concepts used in the thesis are introduced as follows:

- A software engineering **method**<sup>1</sup> provides a technique of “how to” build software. A method is often supported by graphical notation to express the requirements and designs of a software system. The *Structured Methodology* is a traditional approach focuses on specifying and decomposing system functionality from the a . . . on domain.
- An integrated CASE **tool set** or **environment** provides automated or semiautomated support for the system development that is guided by a corresponding method.

---

<sup>1</sup>According to the *Oxford English Dictionary*, methodology is “the science of method”, it is “a treatise or dissertation on method”. We use methodology as the abstract description of a class of methods, e.g., *Structured Methodology* includes structured methods such as Data Flow Diagram (DFD) and Higher Order Software (HOS), etc.

CASE environments operate rather independently. To provide support for multiple CASE environments, Metaview is being developed as an integrated meta CASE facility. The primary advantages of Metaview are:

1. *Definition and Integration of a New Environment*

The meta model and environment definition languages provided in Metaview are very helpful in defining environments. Because Metaview has a mechanism to integrate and maintain a variety of environments, a newly defined environment can be composed with other Metaview created environments rather easily.

2. *Definition of Constraints*

Metaview has a strong capability for defining system constraints during the environment configuration, which makes it possible to check the system consistency and completeness automatically.

3. *Environments Maintenance and Evolution*

A variety of support environments are integrated using Metaview. The system developer can choose one or several environments to support system development. In addition, an environment definer can easily alter or extend an existing environment to match the software development requirements for a particular project.

## 1.2 Basic Concepts of the O-O Method

The 1980s brought a major breakthrough in software engineering with the introduction of O-O approaches to system development [HS91]. These approaches model the real world at many levels of abstraction and thereby support the development of more comprehensible, reusable, and extendable software systems. In addition, O-O methods hold the promise of improving both the quality and the productivity of software development.

The *object-oriented methodology* consists of object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented implementation (OOP). OOA is a stage of decomposing system into objects. A number of classes and their relationships are defined in OOA models. The OOD phase transfers OOA models into design models by grouping the objects into subsystems, and determining the data struc-

ture and algorithms needed to implement each class. OOP is the implementation stage and is not the central focus of this thesis. Object-oriented approaches generally have the basic concepts which have been synthesized from the following references [Bud91, Goo92, Rum91, HS91, KM90]:

- *Object*: An *object* is an entity that has identity, state (through the attributes it contains) and behavior (through the operations or methods it can perform).
- *Class*: A *class* abstracts and groups objects according to their common attributes and behavior. An object is an instance of a class.
- *Inheritance*: A class (the *subclass*) inherits the properties (attributes and operations) of another class (the *superclass*). The relationship between a subclass and its superclass is called *inheritance* or *is-a* relationship. *Multiple inheritance* means that a subclass shares the properties from more than one superclass.
- *Aggregation*: The relationship between an assembly class and its component classes is called *aggregation*. It is also called the *part-of* relationship.

### 1.3 Outline of the Thesis

The thesis is organized as follows. Chapter 2 introduces the basic background of Metaview and illustrates how a specified environment can be defined in Metaview. To gain an understanding of Metaview and its modeling capability, a widely used method, Rumbaugh et al's OMT, is chosen to be prototyped in Metaview. Chapter 3 presents the OMT method and the modeling results. In addition, Metaview's capability to model O-O methods is also critiqued in this chapter to satisfy the first major objective of the thesis. Chapter 4 presents the comparison of several representative O-O methods: OMT, Booch's Method, Fusion Method, and Shlaer-Mellor's Method. On the basis of this comparison and on the experience of OMT modeling, chapter 5 proposes an enhanced method, called the Object-Oriented Modeling (OOM) method. The definition and prototype development of OOM in Metaview completes the second major objective of the thesis. The thesis concludes in chapter 6 with a summary of the major contributions and a discussion of the future research in this area.

## Chapter 2

# Metaview, the Generator of Multiple Environments

Metaview is a system to generate automatically software specification environments used in requirement analysis and design [Bol92, Sor83, McA88, Zhu93]. Because one primary objective of the thesis is to assess how well a metasystem approach can be used to support object-oriented methods, it is important to provide an overview description of Metaview. Section 1 introduces the fundamental concepts in Metaview [Sor91, Fin92a, Fin92b], followed by Section 2 which describes the Metaview architecture [Fin93e]. In Section 3, the modeling schema for software methods and the modeling advantages of the Metaview environment are discussed.

### 2.1 Fundamental Concepts

The meta model in Metaview is called the EARA/GE (*Entity-Aggregate-Relationship-Attribute with Graphical Extension*) model. The *Environment Define Language* (EDL) and *Environment Constraint Language* (ECL) were developed on the basis of EARA/GE model to define a supporting CASE environment.

#### **EARA/GE Model**

The main elements of the EARA are *entity*, *aggregate*, *relationship*, and *attribute*. An *entity* represents an identifiable software object. A *relationship* binds together two or more entities that are somehow interrelated in the modeled software environment.

An *aggregate* [Fin93a] is regarded as a special object representing a collection of entities and relationships. An aggregate is used to show more details or internal structure for a complex entity. For example, in Figure 1 the abstract entity A is associated with the aggregate B, which is composed of two entities C and D, and the relationship R between them. We say that the entity A is exploded into the aggregate B or that the aggregate B is an explosion of the entity A. Each type of

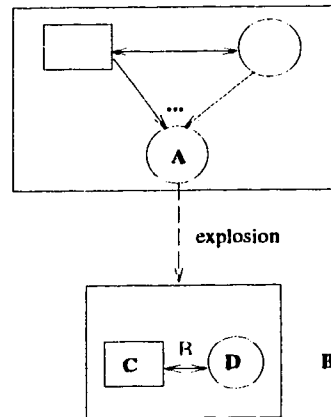


Figure 1: Aggregate in EARA model

entity, relationship and aggregate object can have *attributes*. Each attribute has its own name and value type.

In addition to aggregation, Metaview supports generalization through subtyping. A subtype inherits all attributes (names and types) of its supertype. This leads to more compact and readable environment descriptions.

The Graphical Extension (GE) adds two important aspects to the EARA model. First, graphical counterparts for each of the type sets existing in the EARA Model are defined. In particular, the graphical notation for *icon*, *edge* and *diagram types* are provided as the counterparts of *entity*, *relationship* and *aggregate types*, respectively. The second aspect provides drawing rules (constraints) to ensure for a particular environment the “correctness” of the graphical representation of a software specification.

## EDL and ECL

EDL is based on the EARA metamodel and is used to capture the basic concepts inherent to specification environment [GM93, MG93]. ECL is used to define

constraints formally and thereby support the automatic checking of a specification's consistency and completeness [Fin94b, Fin94c].

## 2.2 Architecture of Metaview

Figure 2 shows how the Metaview system is partitioned into three levels: *the Meta Level*, *the Environment Level*, and *the User Level*.

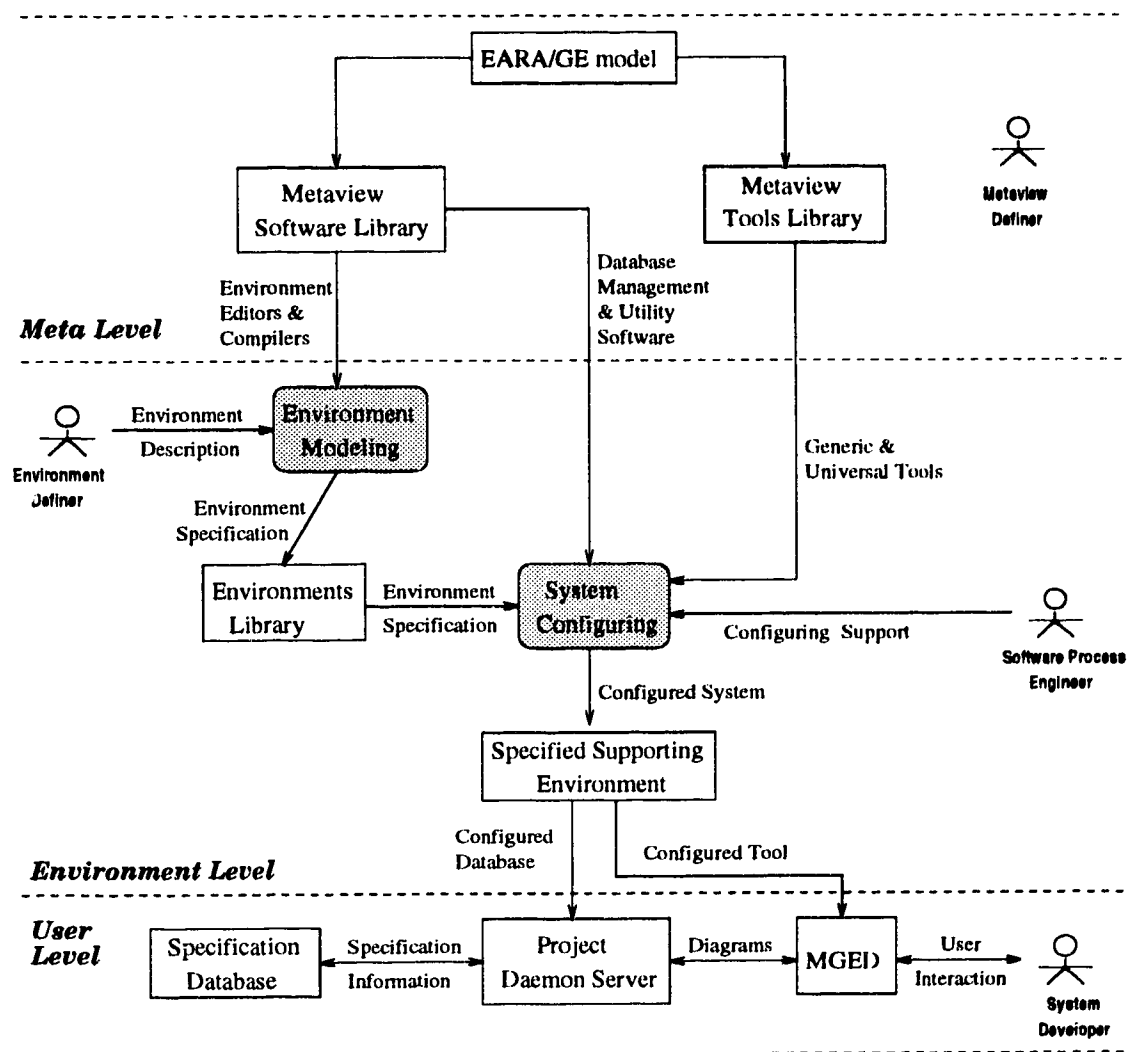


Figure 2: Current prototype configuration of Metaview

### 1. *Meta Level*

At the *Meta Level*, the *Metaview definer* defines the EARA/GE model and then creates two libraries, the *Metaview Tools Library* and *Metaview Software Library*, which provide generic tools and utilities respectively for the other levels. Currently, only the *Metaview Graphical Editor* (MGED) exists in the *Metaview Tools Library*, while the *Metaview Software Library* is richer with the following utilities:

- *Database Engine*  
This prolog-based database management system is used to manage the *Specification Database* at the *User Level*.
- *Project Daemon*  
This server supports the graphical interface to the *system developer* and provides the access to the *Specification Database* at the *User Level* [Fin93d].
- EDL/ECL compilers  
They are used to compile the CASE supporting environments defined in EDL/ECL at the *Environment Level*.

### 2. *Environment Level*

The *Environment Level*, which connects the *Meta Level* and the *User Level*, is the functional level to generate required environments. It is the major level for modeling a software development method (e.g., object-oriented methods).

As shown in Figure 2, the *environment definer* uses the environment modeling software provided at the *Meta Level* to do *environment modeling*. As a result, a formal *environment specification* is generated and stored in the *Environment Library*. An *environment specification* is composed of:

- *Conceptual Tables* which contain the declarations of entity, relationship and aggregate types, as well as their attributes;
- *Graphical Tables* which contain the graphical types (e.g., icon and edge types) that correspond to the conceptual types;
- *Conceptual Constraints and Graphical Constraints* which ensure the consistency and completeness of the CASE environment.

In the fully working Metaview prototype, the conceptual and graphical tables are defined in the EDL language, while their constraints are described in the ECL language.

The *software process engineer* carries out the *system configuration* process to construct a *specified supporting environment*. In the first step of the process, proper generic tools and the environment specification are selected from the *Metaview Tools Library* and the *Environments Library*, respectively. Then the *software process engineer* creates *configured tools* from the selected tools and environment specification, and creates the *configured database* from the environment specification and generic Database Engine.

In summary, two tasks, the *environment modeling* and *system configuring*, are necessary to construct a supporting environment. In the current Metaview prototype, MGED is the primary *configurable tool*. It is supported by the *Project Daemon* which is the general server to manage the *specification database* access for MGED and future generic tools.

### 3. User Level

Prior to using a *specification environment*, the *application system developer* sets up the *Project Daemon Server* and various MGED clients. He/She then edits the software specifications (in the form of diagrams) for the application system using MGED. The specifications are passed by MGED to its *Project Daemon Server*, which processes the diagrams and stores the *specification information* for the application system into the *Specification Database*.

## 2.3 Metaview Modeling Mechanism

### 2.3.1 Method Modeling

To build an environment based on a Metaview development method, two components are created: the *concept\_env* and *graphic\_env* [Fin93c, Fin93b]. Some important features of Metaview, such as its support of aggregation and classification functions, and its support of defining environment constraints, ensure Metaview can generate specified environments easily.



## Concept\_env and Graphic\_env

*Concept\_env* contains the conceptual definition of the environment, which has four basic types of logical components: entities, attributes, relationships and aggregations. *Graphic\_env* defines the graphical extension (GE) to the conceptual definitions. The logical elements defined in *concept\_env* are mapped into corresponding graphical notations defined in *graphic\_env*. For example, in an O-O environment the conceptual entity *class* is represented using the graphical notation of a *box*.

At present, the ECL/EDL compilers are not practical as the ECL part is not fully implemented and tested. A method can be and was modeled using Prolog language statements at the Database Machine level. To make the environment definitions in the thesis easier to read, the corresponding EDL/ECL are used to describe the environment modeling process.

## Aggregation and Classification

One important advantage of Metaview is that it supports an *entity explosion* association called aggregation. This allows the encapsulation of similar entities and their relationships into an abstract entity called aggregate entity. Support for an *aggregation* is built into the system (i.e., supported at the Database Machine level).

Metaview also supports *classification* or *generalization*. If several entities or relationships have the common characteristics, we can define an abstract type and then define its subtype(s) in both *concept\_env* and *graphic\_env* to simplify the modeling. The attributes of abstract terms can be inherited by all the descendants and so it is convenient for the environment developer to generate a software specification environment using inheritance.

## Environment Constraints

Metaview provides two kinds of constraint checking, *completeness checking* and *consistency checking*. *Consistency checking* automatically checks the developer's specification entry and provides error messages when inconsistencies in the specification exist. The wrong modeling operations will result in an inconsistent specification which cannot be performed and saved in the Metaview specification database. *Completeness checking* is a weaker form of checking in the sense that partial defined specifications

are allowed and can be stored in the Metaview specification environment. When a developer finishes part or all of the requirements analysis or design specification, he/she checks whether or not the model is complete. Incompleteness are reported and it is the responsibility of the developer to complete the specification. Completeness and consistency checking are implemented in two places: in *concept\_env* to check the logical modeling, and in *graphic\_env* to check the graphical representations. Constraint checking is a very powerful feature in Metaview.

### 2.3.2 Environment Definition Guidelines

Several environments, such as Data Flow Diagram (DFD) [McA88, Fin94a], Higher-Order Software (HOS) [Sor91], Structure Chart [ST93], ADISSA [Lee92] and Ward-Mellor [Gad94a], have been modeled in Metaview. Based on the experiences gained in modeling these environments, the following guidelines [Gad94a] are adopted to prototype a method:

- *Keep fidelity to the modeled method*

As much as possible, aspects of a method should be modeled in its CASE environment. The original method should not be changed because it is difficult to model some features in Metaview, or because there exist weaknesses within the method.

- *Minimize the complexity of the generated environment*

Classification and aggregation are the major modeling features of Metaview which can decrease the modeling complexity. Therefore, the environment definer should use these abstract concepts whenever possible.

- *Ensure the consistency and completeness*

To ensure the production of high quality specifications, the defined environment should support consistency and completeness checking.

In this thesis, we will attempt to strictly adhere to these guidelines.

## Chapter 3

# Modeling Rumbaugh's OMT in Metaview

After understanding an O-O method, we should be able to construct a prototype environment for the method in Metaview. Among several object-oriented methods, OMT was chosen for modeling primarily because it is a relatively complex and expressive model, and it appears to be the most popular method among those being adopted by the industry.

Figure 3 depicts the architecture of OMT method. Requirements for the system to be developed are accepted as inputs at the OOA phase to do *Object Modeling*, *Dynamic Modeling*, and *Functional Modeling*. *Analysis models* and *relative documents* are outputs of OOA that are stored in a repository. They are used as the inputs to the OOD phase which contains both *System Design* and *Object Design*. The results of OOD, the *design models* and *related documentation*, help generate an *Information System* that is the solution of the problem domain. Iterations exist within the OOA, OOD, and the Implementation phases, within the subphases of the OOA and OOD (e.g. *Object Model* within OOA), and across the OOA, OOD and Implementation phases.

In the following sections, the objectives of OMT modeling in Metaview are discussed first. Then the OMT's OOA and its modeling in Metaview are presented in section 3.2 and 3.3, respectively. Section 3.4 introduces OMT's OOD along with its model description in Metaview. Section 3.5 provides an assessment of the strengths and the limitations of Metaview's capability to model OMT and O-O methods more

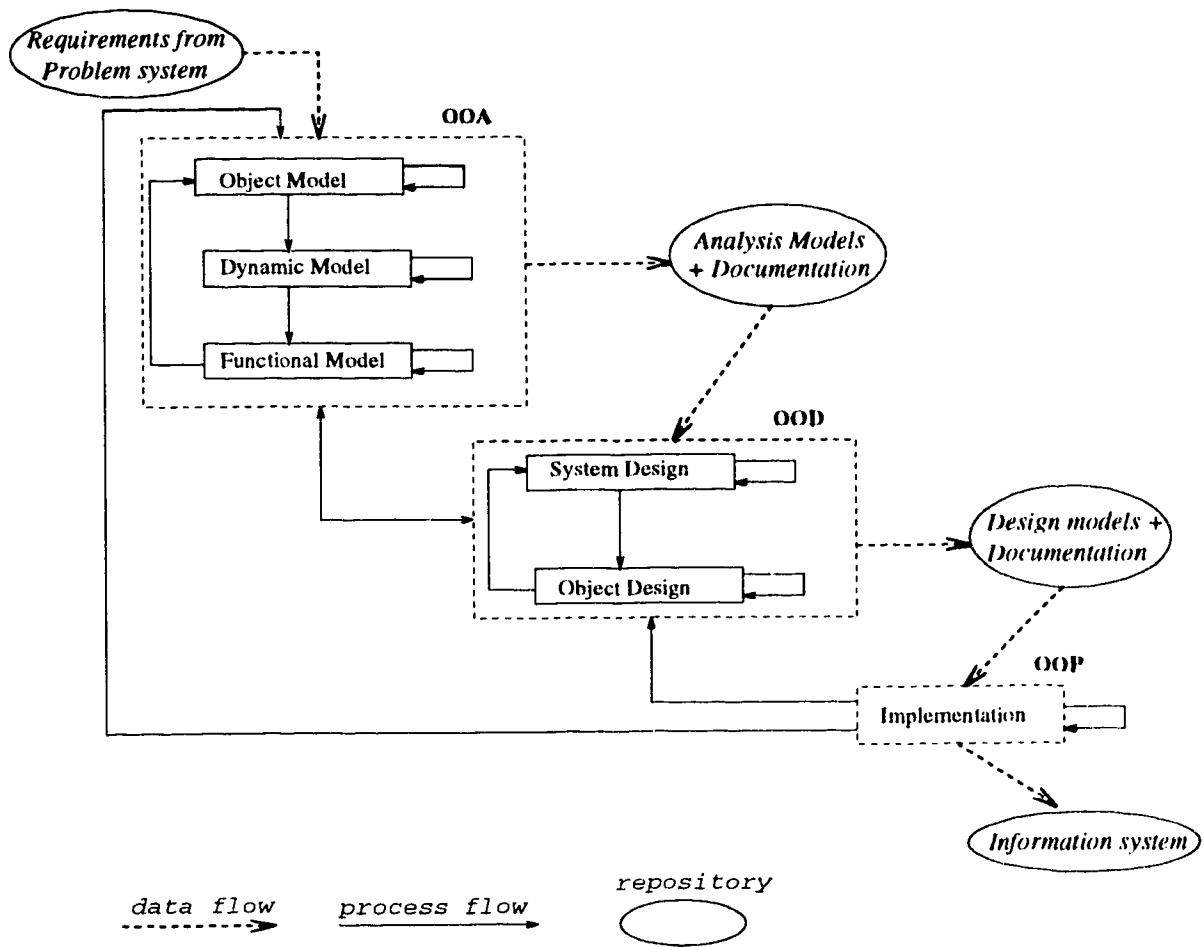


Figure 3: OMT method

generally.

### 3.1 Purpose Of Modeling OMT

#### Why OMT?

Among current object-oriented methods, Rumbaugh *et al's* OMT is among the most popular. Several books and papers [Rum91, Rum94, Gil94, dCF92] that discuss OMT are published and some commercial CASE tools that support OMT are developed. For example, a CASE tool called OMTool [The93] is now provided by the Advanced Concepts Center (ACC) of General Electric to support OMT in both DOS and UNIX environments. These positive results convinced many organizations to adopt this method. Compared with other object-oriented methods, OMT has powerful support for the development of complex systems. OOA, in particular Object Modeling within the OOA, uses many notations to express different situations that may arise in complex application systems. Although some other object-oriented methods, such as Shlaer-Mellor's Method, have similar object modeling techniques, OMT was chosen as the first and representative O-O method for prototyping.

#### Purposes of Modeling OMT

Our effort to prototype the OMT methods in Metaview involves configuring a basic OMT CASE environment. In this thesis, the purposes of modeling OMT are to:

- *Explore the Possibility of Modeling O-O Methods in Metaview*

Metaview is still an exploratory research prototype and thereby there is no guarantee that it can successfully model any arbitrary method. Factors, such as whether the method matches the the fundamental concepts of EARA/GE model, affect method modeling. Although several methods, such as Data Flow Diagram (DFD) and Higher-Order Software (HOS), have been modeled in Metaview, none are object-oriented. Prototyping OMT as a representative O-O method using Metaview allows us to explore the complications of modeling O-O methods in general. In particular, it is valuable to define the O-O concepts, such as *class*, *object*, *inheritance* and *aggregation* etc, in Metaview. This exploration should

uncover the advantages and disadvantages of Metaview and thereby provide valuable insights into how we can improve our metasytem approach.

- *Provide an OMT Environment in Metaview*

One goal of Metaview is to provide multiple-method CASE tools so that developers can choose their favorite method and associated environment to build their application system. OMT is a popular method and so it is worth while to provide an OMT CASE environment in Metaview.

- *Support Other Object-Oriented Modeling*

There exist several popular object-oriented methods and each has strengths and weaknesses. On the basis of an OMT modeling experience, we can compare and critique popular O-O methods, propose an enhanced object-oriented method, and implement it in Metaview. For such enhancements, it is hoped that much of the OMT definition can be reused for the enhanced method.

## 3.2 OMT's OOA

The purpose of OOA is to understand and define the application domain so that a correct design can be constructed. In this section, we introduce how to use Metaview to prototype OOA's three orthogonal models: the Object Model, Dynamic Model, and Functional Model. Each model focuses on a particular aspect, namely, data, sequencing, and operations respectively.

There are two aspects that need to be clarified. First, when we refer to an *object*, we may describe a class or a class instance, or both. In general, it should be clear which concept applies from the context provided. Second, the term *system* is generally used to refer to the application system (or the problem domain) that is being modeled.

In the following discussion, the Object Model, Dynamic Model, and Functional Model are described in succession. Then the environment modeling approach in Metaview is shown for each model.

### 3.2.1 The Object Model

Within the Rumbaugh's OOA, object modeling is the first and the most complex step. The Object Model is used to capture the static structure of an application system via classes/objects and their relationships. Rumbaugh described a series of steps to do object modeling. In order to keep fidelity with OMT, we introduce all the prescribed notation within each step to build the Object Model and then illustrate how to describe the model in Metaview.

#### 1. The Steps to do Object Modeling

##### 1. *Identify the Objects*

Class is the core concept in an object-oriented method. In order to describe a system using OMT, the analyst must first identify all the classes in the application system. Figure 4 illustrates the graphical notation in OMT: *class*, *derived class*, *object instance*, and *derived object instance*. A derived class or object instance is derived from one or more classes or objects, respectively.

##### 2. *Identify the Associations and Aggregation of the Classes*

Association is a special relationship of which there are three kinds of associations in OMT: the *multiplicity association*, *qualified association*, and *ternary association*.

*Multiplicity* specifies how many instances of one class may relate to a single instance of an associated class. It can be divided into five categories: *Exactly one*, *many* (zero or more), *optional* (zero or one), *one or more*, and *numerically specified*. There exists a constraint in the association called *ordering*, which uses the text {*order*} to clarify the order of the “many” side of the association. Figure 5 summarizes the multiplicity of associations and the ordering.

In the process of modeling OMT method, the *one or more* multiplicity and *numerically specified* multiplicity can be modeled by putting *1+* or a specified number in place of the *exactly one* multiplicity.

Multiplicity exists on two sides of an association. For the three most common kinds of multiplicity there exist nine kinds of associations: *many-to-many*, *one-to-one*, *optional-to-optional*, *many-to-one*, *one-to-many*, *many-to-optional*,

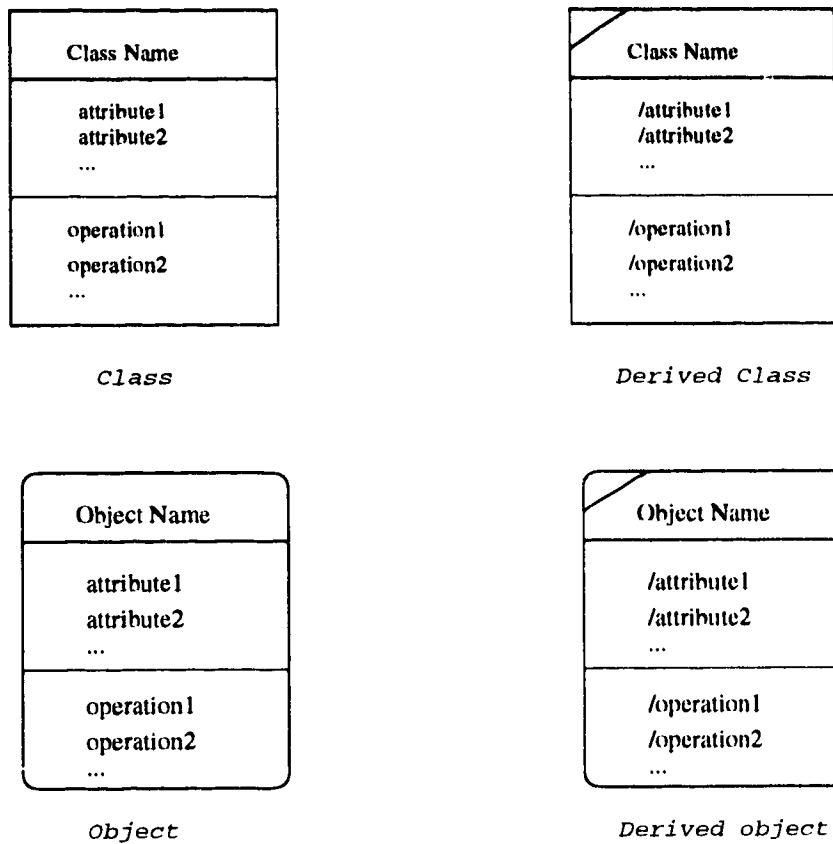


Figure 4: The notation of class and object instance

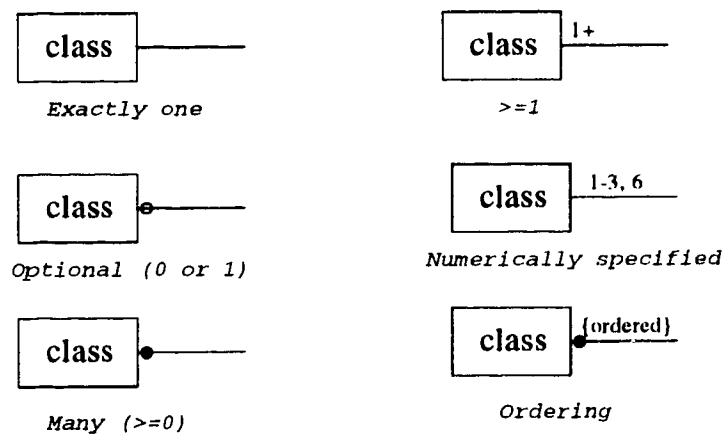
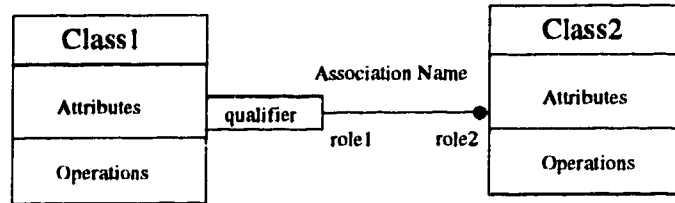


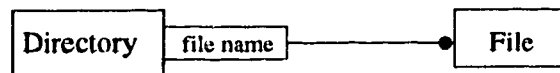
Figure 5: The notation of multiplicity and ordering



*optional-to-many*, *optional-to-one*, and *one-to-optional* association. Besides all these multiple associations, the constraint *ordering* is also modeled in the prototyped OMT environment.



(a) *qualified association*



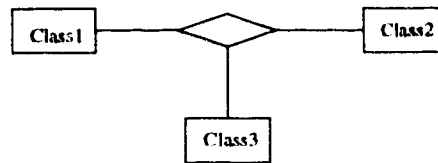
(b) *example*

Figure 6: The notation of qualified association

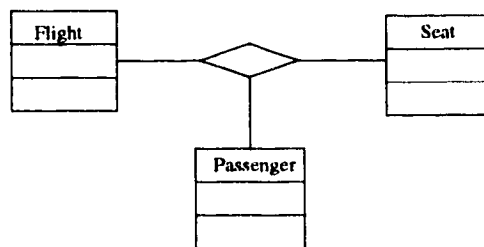
*Qualifier* is a special attribute to reduce the effective multiplicity of an association. A *qualified association* relates two object classes as showed in Figure 6(a). Two kinds of associations can be qualified, the one-to-many and many-to-many. The Figure 6(b) provides an example for the one-to-many association. A qualifier named *file name* is used to specify a unique file which belongs to a directory. That is, a directory plus a file name determines a file.

The general form of an association *ternary* and an example are shown in Figure 7 (a) and (b), respectively. In the example, class *Passenger*, *Flight*, and *Seat* have a ternary association: any *Passenger* who takes *Flight* is assigned a *Seat*.

In this step, an analyst also identifies the *aggregations* (or *is-part-of*) relationship of classes. Figure 8(a) illustrates the general graphical notation. An example of *aggregation* is shown in Figure 8 (b): a *directed graph* is composed of *edges* and *vertices*. The *edges* and *vertices* are components of its assembly class *directed graph*. Recursive aggregation is allowed in OMT. Thus, the components of an assembly class can in turn have their own components.

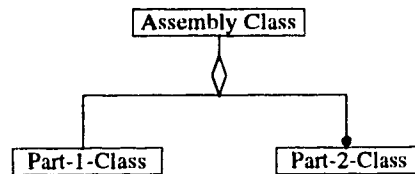


(a) Ternary

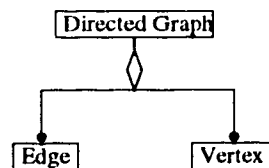


(b) Example for ternary association

Figure 7: The notation of ternary



(a) aggregation



(b) example of aggregation

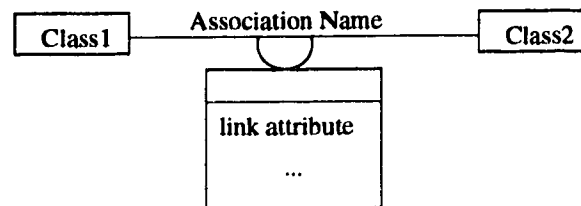
Figure 8: The notation of aggregation

All the graphical notation in this step: nine multiple associations, ternary associations, qualifier, and aggregation, have been modeled in the OMT environment. The implementation details for the prototype are presented later in this section.

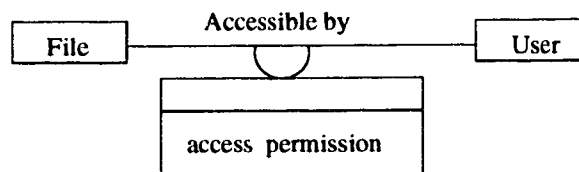
### 3. Identify the Attributes, Operations, and Propagations

The analyst first identifies the attributes of objects, then the *operations* on objects and *propagations* of operations.

The class attribute, object instance attribute, and link attribute are used to express the public interface of a class, instance, and link respectively. In OMT, an analyst can define attributes of the links existing between classes. Figure 9(a) illustrates a general *link-attribute*. In the example shown in the Figure 9(b), the attribute *access permission* is the common property of the association between *File* and *User*. If *access permission* is not defined as a link attribute, it must be modeled as a class, with a ternary association among *User*, *File*, and *access permission*. Using the *link attribute* can reduce the need for *ternary* associations.



(a) link attribute

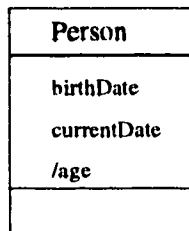


(b) example

Figure 9: The notation of link attribute

*Derived attributes* and *derived operations* are also defined in OMT for the *derived*

*object*. For example, in Figure 10 the class *Person* has the attributes of *currentDate* and *birthDate*. A derived attribute named *age*, which is determined by *currentDate* and *birthDate*, is also declared. A label  $\{age = currentDate - birthDate\}$  is presented in the object modeling diagram to show explicitly the relation between the derived features and original features.



$\{age = currentDate - birthDate\}$

Figure 10: Derived attribute and operation

*Propagation of operations* provides a powerful way to specify a series of operations executed within a group of objects. Assume that a set of associations or aggregations exist among a network of objects. If an operation is applied to some of the objects in the network, the same operation will be automatically applied to the rest of the objects via *operation propagation*.

*Propagation* is indicated with a small arrow and an operation name next to the affected association (or aggregation). The arrow shows the direction of propagation. Figure 11 presents an example showing how the operation *copy* is propagated in three classes, and is connected by two aggregation relationships. A *document* is composed of *paragraphs* which in turn are composed of multiple *characters*. Copying a document causes the copying of paragraphs which in turn causes the copying of characters.

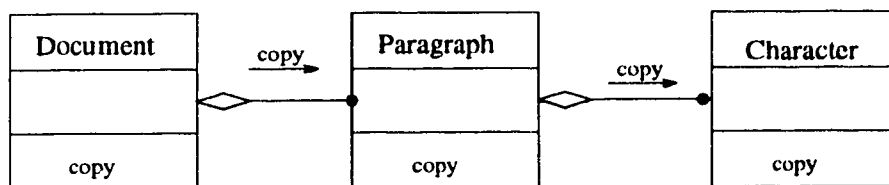
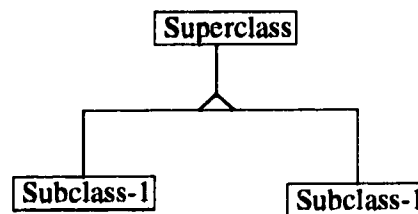


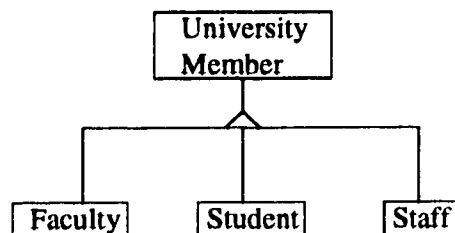
Figure 11: Propagation of operations

#### 4. Identify the Inheritance within the System

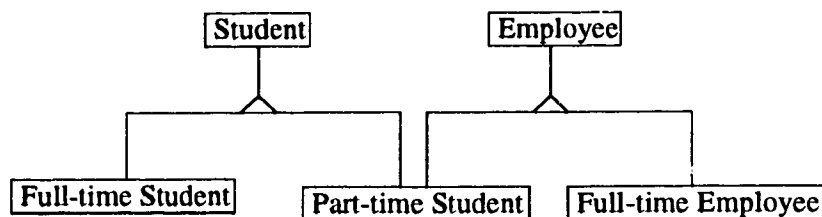
Some classes have similar behaviors. A special relationship called *inheritance* is used to connect a superclass and its subclass(es) which share common behaviors. This relationship, also called *is-a* relationship, can simplify the class description and help to organize concisely the system architecture. For example, in Figure 12 (b), subclasses *faculty*, *student*, and *staff* share their attributes and operations through the superclass *university member*.



(a) Inheritance



(b) Example of inheritance



(c) Example of multiple inheritance

Figure 12: The notation of inheritance(generalization)

OMT also allows *multiple inheritance*, that is, a subclass can inherit operations and attributes from multiple parents. In Figure 12 (c) for example, *Part-time*

*Student* has two super classes: *Employee* and *Student*.

## 5. Constraints

*Constraints* exist between objects, attributes, links, and associations. Figure 13 depicts three constraint notations:

- *Object's Constraints*

A constraint restricts the value scope of an object's attribute. For example, in Figure 13(a) the balance of a bank account is constrained to be not less than zero. Also Figure 10 shows the constraint to the derived attribute *age*. This kind of constraint is expressed by putting a label under the object box.

- *Constraints On Links*

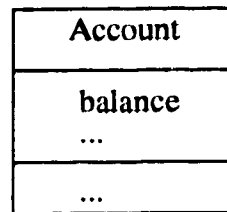
This type of constraint includes the *qualifier*, *ordering*, *multiplicity*, *link attribute* and *label*, which are expressed by placing them near their link lines. Figure 13(b) illustrates an example: both the qualifier constraint *filename* and ordering constraint *ordered* are shown to apply between the classes *Directory* and *File*.

- *General Constraints*

Figure 13(c) depicts how a special relationship between two associations or two classes can be expressed by drawing a dotted line between the two relationship edges. This figure illustrates the general constraint *subset* that exists between the many-to-one association *member-of* and the many-to-one association *manager-of*.

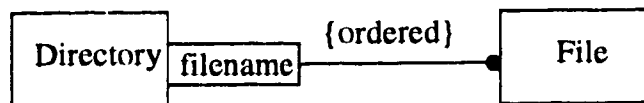
## 6. Iterate and Refine the Model

It is almost impossible for the analyst to describe correctly the static system using classes and their relationships the first time. User requirements are often misunderstood or changed during the period of an analyst's object modeling. An analyst may be forced to revise the model several times to clarify object (attributes and operations) definitions, repair errors, add details, and correctly capture the constraints within the system. It is also important for the analyst to refine the object model to add inheritance, thereby decreasing the complexity of the system by deleting redundant classes and relationships.

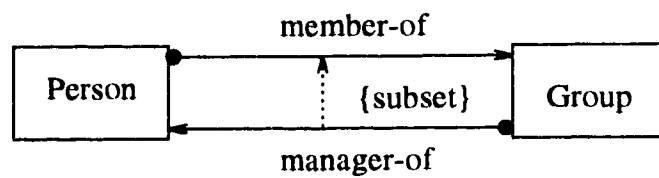


(balance ≥ 0)

(a) Constraints on objects and attributes



(b) Constraints on links



(c) Subset constraint between associations

Figure 13: The notation of constraints

## 7. Group Classes into Modules

Some classes may have a rather close connection and so they can be grouped into a module. In this way, the analyst can partition an object model into manageable pieces, which helps in providing a clear system description.

### 3.2.2 The Dynamic Model

After gaining an understanding of the system by examining its static structure and building an object model, an analyst examines changes to the objects and their relationships over time, and expresses these changes in a *Dynamic Model*.

Traditional finite state machines (FSM), represented as the *state diagrams*, are used to create the *Dynamic Model*. The pattern of events, states, and state transitions for a given class can be abstracted and represented as a *state diagram*.

#### 1. The Dynamic Modeling Steps

##### 1. Prepare Scenarios and Identify the Events

At the beginning of dynamic modeling, an analyst builds scenarios to understand the system behavior. A scenario is a sequence of events that cause the state changes due to external stimuli also called *events*. An event trace is prepared for each scenario to provide a detailed understanding of the system's behavior.

##### 2. Build State Diagrams

For each object class in the Object Model, a *state diagram* is built to show its dynamic behavior caused by the events the object receives and sends. The transformation from the *Object Model* to *Dynamic Model* is relatively easy. All the elements in the *Dynamic Model*, such as *event*, *state*, *action* and *activity*, are used to build *state diagrams*. Such a construction is rather complex and so is discussed further in the following subsection.

##### 3. Checking the Consistency of the Events

When the state diagram for each class is complete, it is necessary to check its consistency and completeness. For example, a state without either a predecessor or successor is incomplete. In our OMT prototyping environment, many



consistency and completeness checks have been implemented, many of these are presented in the Appendix.

## 2. How to Build a State Diagram

In the *Dynamic Model*, *state* is the most important concept. Any object has three types of states: *initial state*, which is entered on creation of an object, *final state*, which implies destruction of the object, and *intermediate state*. Over time, the object is stimulated via events, which results in a series of state changes. A change in an object's state is called a *transition*. Transitions of an object can be modeled as arcs that emanate from an the initial state node, connect all intermediate state nodes, and terminate at the final state nodes.

Some optional terms: *condition*, *activity*, and *action* are defined in *Dynamic Model*. Figure 14 shows how they are depicted as graphical notation. *Condition* can be used as a guard on transitions and is depicted as a boolean expression in brackets following the event name.

*Operation* is a behavioral description of an object which specifies what the object does in response to events. There are two kinds of operations: *activity* and *action*. An *activity* is an operation that takes time to complete. An activity starts on entry to the state and stops on exit.

An *action* is an instantaneous operation associated with an event. *Entry/Exit actions* are two special actions that take place when a state is entered/left.

As shown in Figure 14, a *state* is drawn as a round cornered rectangle which contains the state name. An *initial state* is shown as a solid circle and a *final state* as a bull's eye. An *event* is represented as an arrow from the source state to the target state with event name placed near the arrow. The arrow represents the *transition* caused by an event. An event name may optionally be followed by one or more attributes within parentheses. If applicable, a *condition* may be listed within square brackets after an event name. An *activity* is indicated within a state box by the keyword *do:* followed by the name or description of the activity. An *action* is indicated on a transition following the event name delimited by the '/' character.

Some advanced dynamic modeling concepts are:

- *Internal Actions*

An event can cause an action to be performed without a state change. Such an action is an *internal action*. It is expressed within the state box by an event name followed by the internal action name. As an example shown in Figure 14, the *entry/entryAction* is an internal action within *state1*.

- *Sending Events*

Events can also be sent from one object to another. The dotted line from a transition to an object indicates that an event is sent to the object when the transition is triggered. Figure 14 depicts an example of *sending events* to the object *class1*.

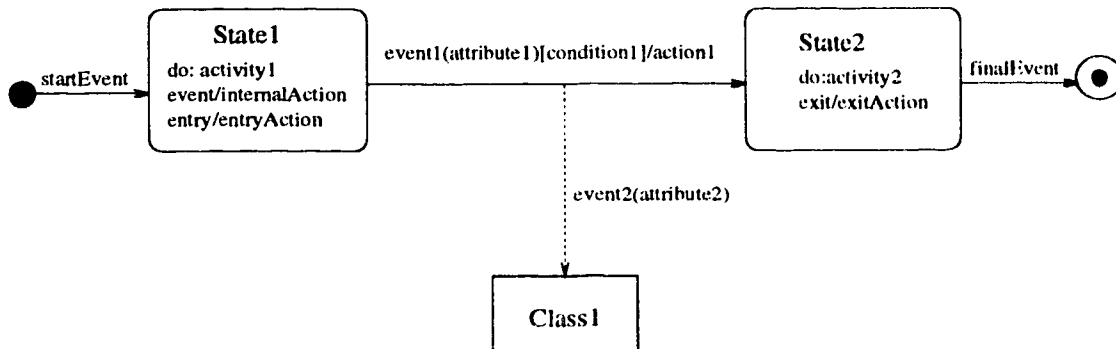


Figure 14: State transition and sending events

- *Nesting State Diagrams*

In a state diagram, states and their activities can be organized into generalization hierarchies to improve understandability and share structure and behavior. An activity in a state can be expanded as a lower-level state diagram, where each state represents one step of the activity.

States can have substates that inherit the transitions of their superstates, just as classes have subclasses that inherit the attributes and operations of their superclass. Any transition that applies to a state also applies to all its substates, unless overridden by another transition on the substate.

We implement substate structures using aggregation in Metaview.

- *Concurrency*

When an object can be partitioned into subsets of attributes, the state of the object is composed of substates. The same event can cause transitions of various

substates, and therefore concurrency arises within an object. These concurrent subactivities must be completed before going to the next state. Figure 15 shows the graphical expression of this concept. After receiving *event1*, the object transforms its state from *State1* to *State2* which is composed of *substate2.1* and *substate2.2*. The subevents of *event2* is partitioned into *event2.1* and *event2.2*, which cause the concurrent subactivities of the object. When the concurrency completes, the object transforms its state to *State3*.

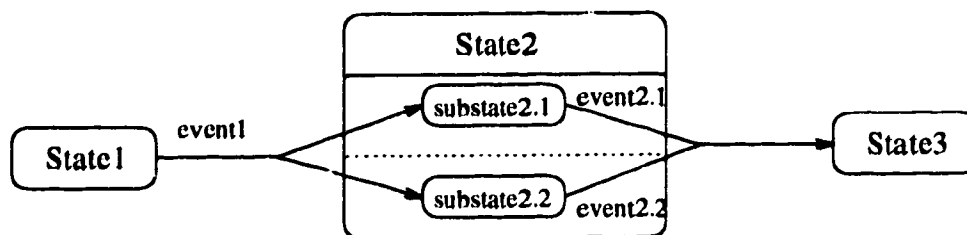


Figure 15: Synchronization of concurrent activities

### 3.2.3 The Functional Model

Using aspects of the Object Model and Dynamic Model as a basis, *Functional Modeling* is the last phase in OOA. It describes how values are computed without regard for sequencing or object structure. A Data Flow Diagram (DFD) is used to construct Functional Models. It contains *processes* to transform data, *dataflows* to move data, *actors* (terminators) to produce and consume data, and the *data stores* to store data passively.

Although the *Functional Model* focuses on the analysis of system functions and is not closely related to the other two models, there still exist some connections between the *Functional Model* and the other two models. For example, the processes in DFD correspond to activities or actions in the *State Model*, and the dataflows in DFD correspond to objects or attribute values in an *Object Model*.

#### 1. Steps to Build Functional Model

##### 1. Identify the Input and Output Values

Input and output values are parameters of events between the systems and the outside world.

## 2. *Build Data Flow Diagrams*

Building a data flow diagram is a complex activity and so is introduced in next subsection.

## 3. *Describe Functions*

A description for each function should be written to explain what the function does.

## 4. *Identify Constraints*

In a system under development, many semantic constraints exist. For example, in a banking system, a constraint might be that client's account balance may never be negative when the client wants to withdraw cash from the account. The analyst identifies constraints in the *functional model* as preconditions and postconditions of the functions.

## 5. *Optimizing the Functional Diagrams*

At the end of functional modeling, an analyst optimizes the analysis models by decreasing the communications between sites, where a site can be a data store, an actor, or a processor.

# 2. **Build Data Flow Diagrams**

The Functional Model consists of multiple *data flow diagrams* (DFD) which show what output values are derived from input values, without specifying how or when they are computed. A DFD shows the flow of data from external inputs, through operations and internal data stores, to external outputs. Figure 16 shows the different elements in a Data Flow Diagram.

A *process* is drawn as an ellipse containing a process's name. An *actor* is drawn as a rectangle containing an actor's name.

A *data flow* represents an intermediate data value that can be changed by a process or actor. A data flow is drawn as an arrow between the producer and the consumer of the data value. Sometimes a composite data value is split into its components, each of which goes to a different process. For example, in Figure 16 the data value *data4* is split into *data4.1* and *data4.2*.

A *data store* is a passive object that contains data for later access. It does not process any operations but merely responds to requests to store and access data.

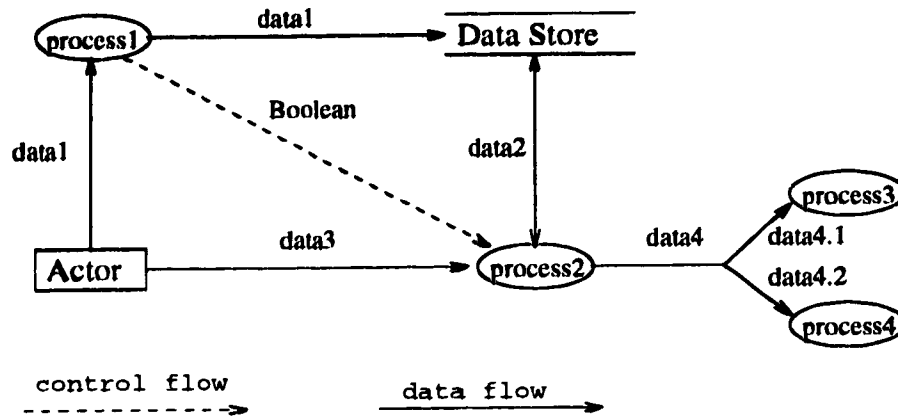


Figure 16: The basic notation of Data Flow Diagram

Some advanced concepts are as follows:

- *Nested Data Flow Diagrams*

A process can be expanded into another data flow diagram. Each input and output of the process is an input and output of the new diagram.

- *Control Flow*

A control flow is not an input data value to the process, rather it is a Boolean value that affects whether a process is to be invoked or not. A control flow is shown as a dotted line from a process producing a Boolean value to the process being controlled. In Figure 16, *process1* sends a control flow to *process2* to decide whether *process2* can take action or not.

### 3.2.4 OOA Models Integrity

In OOA, the *Object Model* describes real-world object classes and their relationships; the *Dynamic Model* describes the sequences of the class states in its life cycle; and the *Functional Model* uses Data Flow Diagrams to describe what data flow and is processed within the system. These three models are used to address different aspects of the modeled system.

Although iterative analysis is performed within object modeling, dynamic modeling, or functional modeling, inconsistencies and imbalances may still exist cross these three models. After defining the three models in OOA, the analyst should iterate

between the different stages to produce a consistent and complete analysis for the OOD activity.

### 3.3 Modeling OMT's OOA

The three models of OMT's OOA are modeled in succession using Metaview. To simplify the modeling, an abstract entity called **universal\_entity** and an abstract relationship called **universal\_relationship** are defined as:

```
ENTITY_TYPE universal_entity GENERIC
  ATTRIBUTES (description: text);
RELATIONSHIP_TYPE universal_relationship GENERIC
  ATTRIBUTES (description: text);
```

#### The Object Model

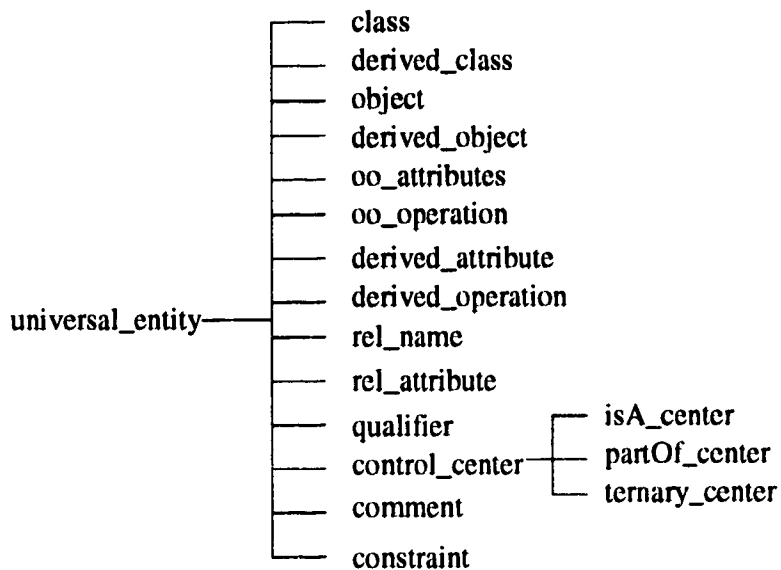


Figure 17: Entities in Object Model

#### Entities:

Figure 17 summarizes the OMT *Object Model's* entities modeled in Metaview. Figure 18 illustrates the entity notations except those entities modeled in the form of text,

such as **oo\_attribute** and **oo\_operation**. The **oo\_attribute** and **oo\_operation** are the attributes and operations of a **class** or **object**. The **derived\_attribute** and **derived\_operation** are used to express the derived attributes or operations of a **derived\_class** or **derived\_object**. There are various types of relationship between classes, objects, **derived\_classes** and **derived\_objects**. The name of a relationship is modeled by the entity **rel\_name**. The **rel\_attribute** is used to show the link attribute (refer to Figure 9). OMT supports some special associations, such as qualified association, ternary association, aggregation and inheritance. The entity **qualifier** is created to show the qualified association. The entity **control\_center** and its three subtyping entities, called **ternary\_center**, **partOf\_center** and **isA\_center**, are used to model the rest three special associations. Using **control\_center** as the super entity type is for increasing the OMT's modeling efficiency. The **comment** and **constraint** entity types are useful to express the label or constraint in Object Model diagrams.

Defining entities in EDL is quite similar and therefore only the examples of defining the **class** and **attribute** entities are presented:

#### ENTITY\_TYPE

```
class GENERIC IS_A universal_entity,
oo_attribute GENERIC IS_A universal_entity;
```

#### Relationships:

To efficiently define OMT environment, the OMT Object Model's relationships can be categorized by three types: **simple relationships**, **complex relationships** and **indirect relationship**. Figure 19 illustrates different relationship types within *Object Model*. The **simple relationships** are those that can be modeled directly and simply. A **complex relationship** comprises a relationship and a set of its subtyping relationships. For example, six multiplicity associations (**many\_to\_many\_associate**, **many\_to\_one\_associate**, etc) are subtyping relationships of **associate**. The relationship type **associate** and its subtyping relationships are modeled together as a **complex relationship**. Some relationship types have semantic constraints and are difficult to be modeled directly. They are replaced by the **indirect relationships** which are modeled as **simple** or **complex relationships**.

- *Simple Relationships*

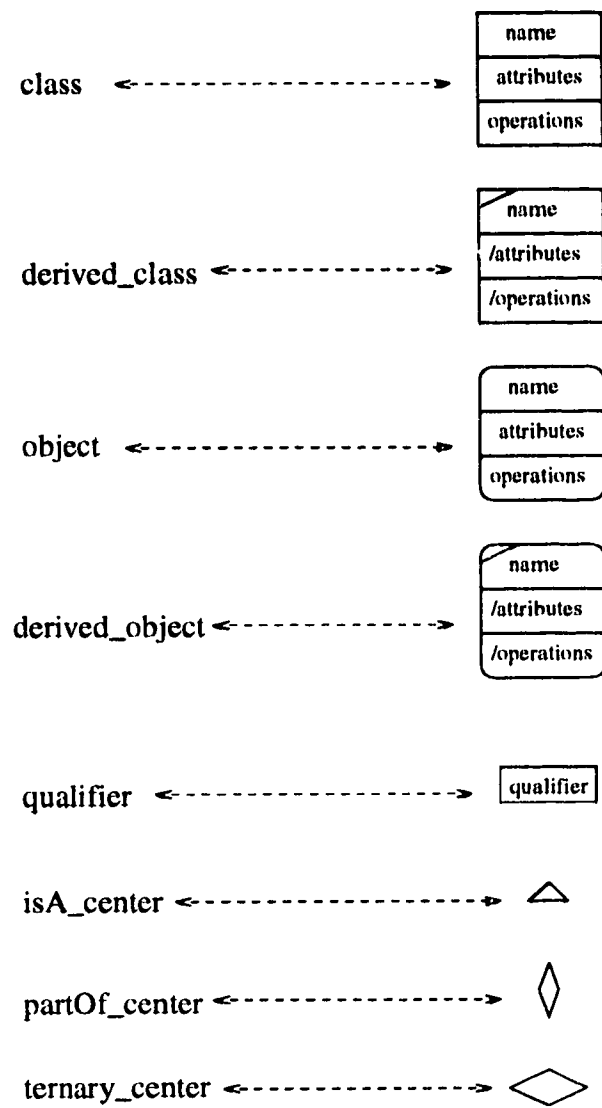


Figure 18: Some entity notations in Object Model



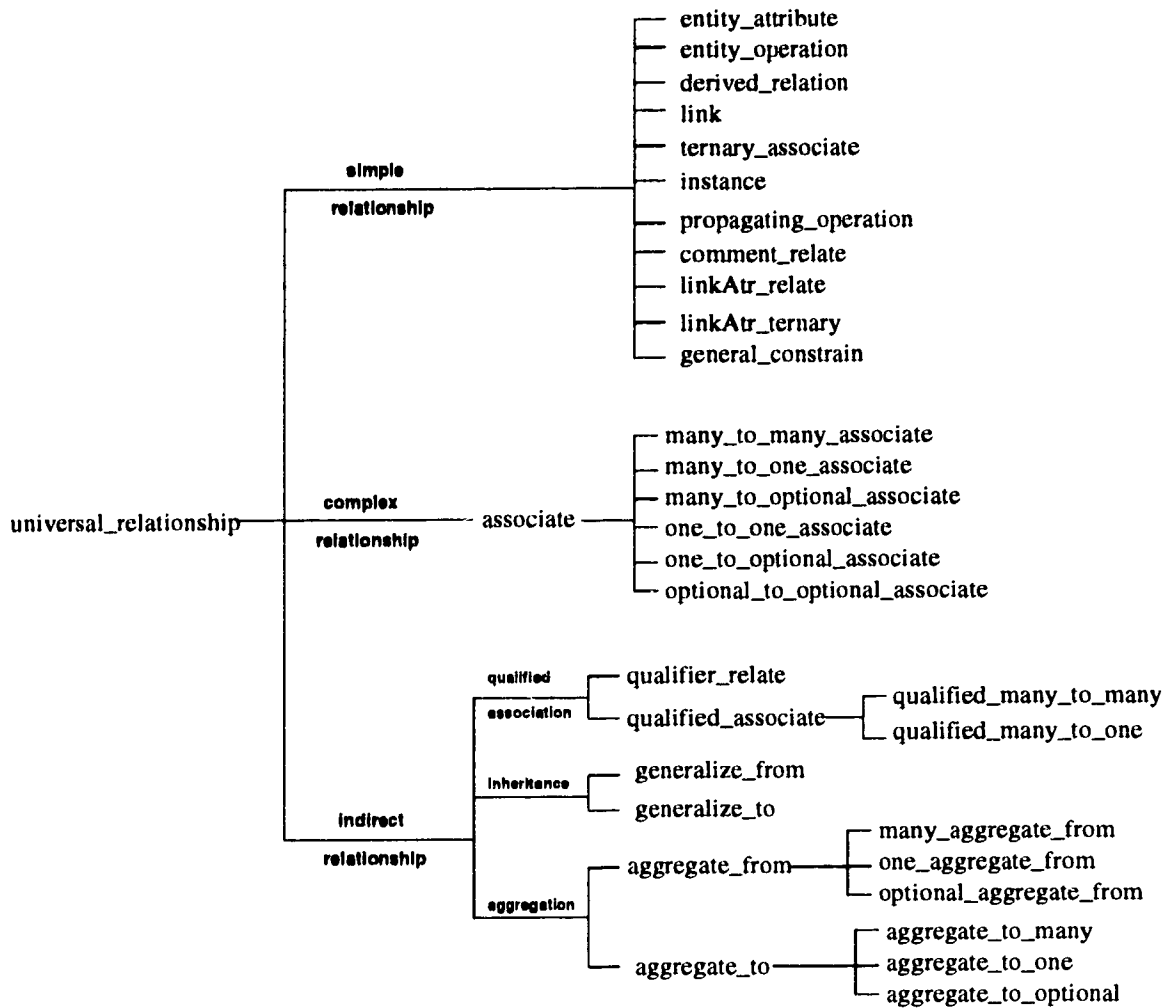


Figure 19: Relationship notation in Object Model

The entity type **class**, **derived\_class**, **object**, **derived\_object** or **rel\_attribute** all have their own attributes and operations. The relationship types **entity\_attribute** and **entity\_operation** define these two relationship types, respectively. As they are quite similar, only the relationship type **entity\_attribute** is presented. The following definition for **entity\_attribute** defines the relationship between a **class**, **derived\_class**, **object**, **derived\_object** or **rel\_attribute**, and an **attribute**. The entity **rel\_attribute** is the *link attribute* (refer to Figure 9) which depicts the attributes of an association or a ternary association. The **rel\_attribute** may have its own attribute.

#### RELATIONSHIP\_TYPE

```
entity_attribute GENERIC IS_A universal_relationship
  ROLES(entity_body, attribute_name)
  PARTICIPANTS
    (class|derived_class|object|
     derived_object|rel_attribute, oo_attribute);
```

The relationship type **derived\_relation** shows the relationship between a class (or object) and a **derived\_class** (or **derived\_object**). **link** is the relationship type between different objects. The **ternary\_relate** relationship type connects a class entity and its **ternary\_center**, which is the connecting node of three participating classes. The relationship type **instance** expresses the relationship between a class and its object instance. OMT supports the propagation of operations (shown in Figure 11) through the relationship type **propagating\_operation**. Some entity types need a constraint as a label to describe them. Figure 10 (a) illustrates an example of the constraint **balance**  $\geq$  0 to the object **account**. The relationship type **comment\_relate** ensures an entity type **comment** as a label to attach to its specified entity. At first, it is considered to model **comment** as the attribute of any specified entity. However, in current Metaview prototype, the display location of an attribute is defined by EDL and so the location of attribute **comment** can not be adjusted. To avoid it, the **comment** is defined as an entity in the OMT support environment to constrain the entity types **class**, **derived\_class**, **object**, **derived\_object**, **isA\_center**, **partOf\_center**, or **ternary\_center** using the relationship type **comment\_relate**. The **linkAttr\_relate** and **linkAttr\_ternary** implements the relationships be-

tween a **rel\_attribute** and its **association** or **ternary association**, respectively. Because a constraint, as shown in Figure 13(c), can exist between different associations of two classes, the relationship type **general\_constraint** is used to express this case. These relationship types are modeled in EDL as follows:

#### RELATIONSHIP\_TYPE

```

derived_relation GENERIC IS_A universal_relationship
    ROLES(class1, relation, class2)
    PARTICIPANTS(class, rel_name, derived_class)
        (object, rel_name, derived_object);
link GENERIC IS_A universal_relationship
    ROLES(object1, relation, object2)
    PARTICIPANTS(object, rel_name, object);
ternary_association GENERIC IS_A universal_relationship
    ROLES(class, ternary_structure)
    PARTICIPANTS(class, ternary_center);
instance generic is_a UNIVERSAL_RELATIONSHIP
    ROLES(instant, relation, class)
    PARTICIPANTS(object, rel_name, class);
propagating_operation GENERIC IS_A universal_relationship
    ROLES(sender, operation_name, recipient)
    PARTICIPANTS(class, oo_operation, class);
comment_relate GENERIC IS_A universal_relationship
    ROLES(attaching, attached)
    PARTICIPANTS(comment, class|object|derived_class|
        derived_object|isA_center|
        partOf_center|ternary_center);
linkAtr_relate GENERIC IS_A universal_relationship
    ROLES(attaching, attached)
    PARTICIPANTS(rel_attribute, rel_name);
linkAtr_ternary GENERIC IS_A universal_relationship
    ROLES(attaching, attached)
    PARTICIPANTS(rel_attribute, ternary_center);
general_constraint GENERIC IS_A universal_relationship

```

```

ROLES(entity_from, constraint_name, entity_to)
PARTICIPANTS(class, constraint, class)
            (rel_name, constraint, rel_name);

```

- *Complex Relationships*

**associate** is a relationship type between two classes. It has six<sup>1</sup> subtyping relationship types: **many\_to\_many\_associate**, **many\_to\_one\_associate**, **many\_to\_optional\_associate**, **one\_to\_one\_associate**, **one\_to\_optional\_associate** and **optional\_to\_optional\_associate**.

**RELATIONSHIP\_TYPE**

```

associate GENERIC IS_A universal_relationship
    ROLES(class1, relation, class2)
    PARTICIPANTS(class, rel_name, class)
    ATTRIBUTES (role1_name: string, role2_name: string);

```

```

many_to_many_associate IS_A associate;
many_to_one_associate IS_A associate;
many_to_optional_associate IS_A associate;
one_to_one_associate IS_A associate;
one_to_optional_associate IS_A associate;
optional_to_optional_associate IS_A associate;

```

- *Indirect Relationships*

There are three types of **indirect relationships** in the prototype of OMT's *Object Model*: the **qualifier\_relate** and **qualified\_association** to model *qualified association* between classes; the **generalize\_from** and **generalize\_to** to model the *inheritance* between classes; and **aggregate\_from** and **aggregate\_to** to model the *aggregation* between classes.

### 1. *Qualified association*

---

<sup>1</sup>The order of choosing two classes determines different associations between classes although the same relationship type is used. The relationship type **many\_to\_one\_associate** can model both **many\_to\_one\_associate** and **one\_to\_many\_associate**. It is similar to the relationship types **many\_to\_optional\_associate** and **one\_to\_optional\_associate**. Therefore, only six relationship types are defined in OMT support environment for nine associations in Rumbaugh's OMT. The relationship type **associate** is used to improve the OMT's modeling efficiency.

As shown in Figure 6, the entity type **qualifier** sticks to the left class and is linked to the right class. There exists two types of relationships. One is called **qualifier\_relate**, which connects the **qualifier** and its adjacent class; the other is called **qualified\_associate**, which connects **qualifier** and another class by a link. **qualified\_associate** has two subtyping relationship types **qualified\_many\_to\_many** and **qualified\_many\_to\_one** to depict the multiplicity role of the linked class. The relationships are modeled as:

#### RELATIONSHIP\_TYPE

```

qualifier_relate GENERIC IS_A universal_relationship
    ROLES(attaching, attached)
    PARTICIPANTS(qualifier, class);

qualified_association GENERIC IS_A universal_relationship,
    ROLES(attaching, relation, class)
    PARTICIPANTS(qualifier, rel_name, class)
    ATTRIBUTES (role1_name: string, role2_name: string);
qualified_many_to_many IS_A associate;
qualified_many_to_one IS_A associate;

```

## 2. Inheritance

In OMT, **inheritance** between two classes is expressed by connecting both classes to a common entity type **isA\_center** (refer to Figure 12 (a) ). If the association between any class to the **isA\_center** is defined as **inheritance**, it is difficult to distinguish the superclass and subclass. Therefore two indirect relationships, the **generalize\_from** and **generalize\_to**, are used to implement the semantic relationship **inheritance**.

#### RELATIONSHIP\_TYPE

```

generalize_from GENERIC IS_A universal_relationship,
    ROLES(superclass, isA_structure)
    PARTICIPANTS(class, isA_center)
    ATTRIBUTES (role1_name: string, role2_name: string);

generalize_to GENERIC IS_A universal_relationship,

```

```

ROLES(isA_structure, subclass)
PARTICIPANTS(isA_center, class)
ATTRIBUTES (role1_name: string, role2_name: string);

```

### 3. Aggregation

Similar to the inheritance relationship, the indirect relationship types **aggregate\_from** and **aggregate\_to** are used to distinguish the assembly class and its components. In OMT's *Object Model*, the **aggregate\_from** has three subtyping relationships: **many\_aggregate\_from**, **one\_aggregate\_from** and **optional\_aggregate\_from**. These three relationship types show the multiplicity role of the superclass; the **aggregate\_to** also has three subtyping relationships: **aggregate\_to\_many**, **aggregate\_to\_one** and **aggregate\_to\_optional**. They are modeled as follows:

```

RELATIONSHIP_TYPE
aggregate_from GENERIC IS_A universal_relationship
    ROLES(class, partOf_structure)
    PARTICIPANTS(class, partOf_center);

many_aggregate_from IS_A aggregate_from;
one_aggregate_from IS_A aggregate_from;
optional_aggregate_from IS_A aggregate_from;

aggregate_to GENERIC IS_A universal_relationship
    ROLES(partOf_structure, component)
    PARTICIPANTS(partOf_center, class);

aggregate_to_many IS_A aggregate_to;
aggregate_to_one IS_A aggregate_to;
aggregate_to_optional IS_A aggregate_to;

```

### Aggregates:

The Object Model contains one aggregate **object\_model**, which includes all the entities and relationships defined in the *Object Model*.

```

AGGREGATE_TYPE object_model
COMPONENTS (ALL);

```

## The Dynamic Model

A *Dynamic Model* is used to analyze the state changes of an object. Figure 20 lists all the notation in the *Dynamic Model* except the entities modeled in the form of text, such as a state *activity* or *action*. The first three notations in Figure 20 illustrate different types of states the analyzed object has during its lifetime. The remaining two notations show two types of transitions. **send\_event** indicates how the currently analyzed object sends an event to an external object. Only event name is shown on the arrow of the **send\_event** relationship type. **control\_flow** exhibits how the object state is changed from one state to another with an event sent out to stimulate the destination state. The event name should be placed on the state transition arrow, optionally followed by the event attribute, the condition of transition, and the event action. Figure 14 illustrates how **send\_event** and **control\_flow** works.

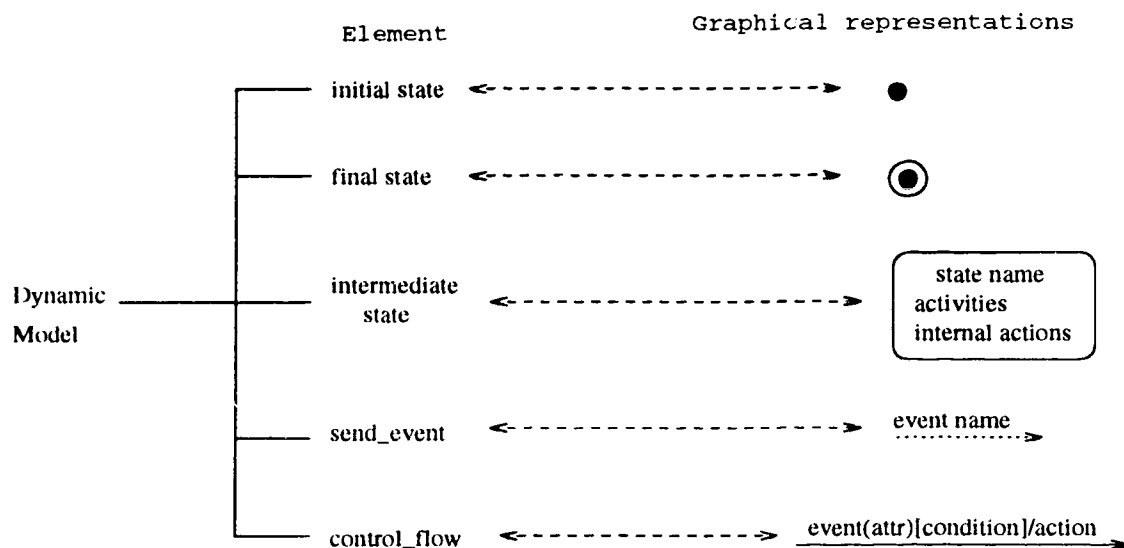


Figure 20: Notation of Dynamic Model

### Entities:

The entity **state** which is classified into **initial\_state**, **intermediate\_state** and **final\_state** are modeled in EDL as:

**ENTITY\_TYPE**

```

state GENERIC IS_A universal_entity;
initial_state IS_A state;
intermediate_state IS_A state;
final_state IS_A state;

```

The entity type **transition\_name**, which display event and the optional three (*event attribute*, *event condition* and *action*), is created to model the **control\_flow**. The value of the entity type **transition\_name** identifies the event, whereas the *event attribute*, *event condition* and *action* are modeled as the attributes of **transition\_name**. In Metaview, the entity type's attributes are displayed on the screen on condition that they are assigned values. This makes it possible to display the *event attribute*, *event condition* and *action* optionally.

The state activities and internal actions are expressed by the entity types **activity** and **action**, respectively. The entity types **transition\_name**, **activity** and **action** are modeled as:

**ENTITY\_TYPE**

```

transition_name GENERIC IS_A universal_entity
  ATTRIBUTES (action_name: string, event_condition: string,
              event_attribute: text);
activity GENERIC IS_A universal_entity;
action GENERIC IS_A universal_entity;

```

**Relationships:**

Relationship type **control\_flow** represents how one state is transferred to another one. Three control\_flows exist: from an **initial\_state** to an **intermediate\_state**, and from an **intermediate\_state** to an **intermediate\_state** or **final\_state**.

**send\_event** shows how a event is sent from an intermediate state of the analyzed object to an external class. Because a intermediate state has activities and internal actions, two relationship types called **state\_activity** and **state\_action** were created in the OMT support environment. The relationships are modeled as:

**RELATIONSHIP\_TYPE**

```

control_flow GENERIC IS_A universal_relationship

```



```

ROLES(source, flow, destination)
PARTICIPANTS
  (initial_state, transition_name, intermediate_state)
  (intermediate_state, transition_name, intermediate_state|final_state);

send_event GENERIC IS_A universal_relationship
  ROLES(source, flow, destination)
  % rel_name and class have been modeled in the Object Diagram
  PARTICIPANTS (transition_name, rel_name, class);

state_activity GENERIC IS_A universal_relationship
  ROLES(state, inActivity)
  PARTICIPANTS (intermediate_state, activity);

state_action GENERIC IS_A universal_relationship
  ROLES(state, inAction)
  PARTICIPANTS (intermediate_state, action);

```

The relationship type **control\_flow** can also be used to create the state synchronization of concurrent activities depicted in Figure 15. In the Metaview Graphical Editor (MGED), the user can draw poly-line by choosing several anchor points. Hence, the concurrency of multiple **control\_flows** can be drawn by providing shared line part.

### Aggregates:

The Dynamic Model's aggregate type *state\_model* contains every element type in the model. Because OMT supports nested state diagrams, the entity type **intermediate\_state** is modeled to link the aggregate type *state\_model* to implement the nesting.

```

AGGREGATE_TYPE state_model
  COMPONENTS (transition_name, activity, action, state, %ENTITIES
    % reused from object_model
    class, rel_name,
    %relationships
    control_flow, send_event,

```

```
state_activity, state_action);
```

```
ENTITY_TYPE intermediate_state IS_A state
BECOMES state_model;
```

## The Functional Model

Figure 21 depicts the notation used in the *Functional Model*.

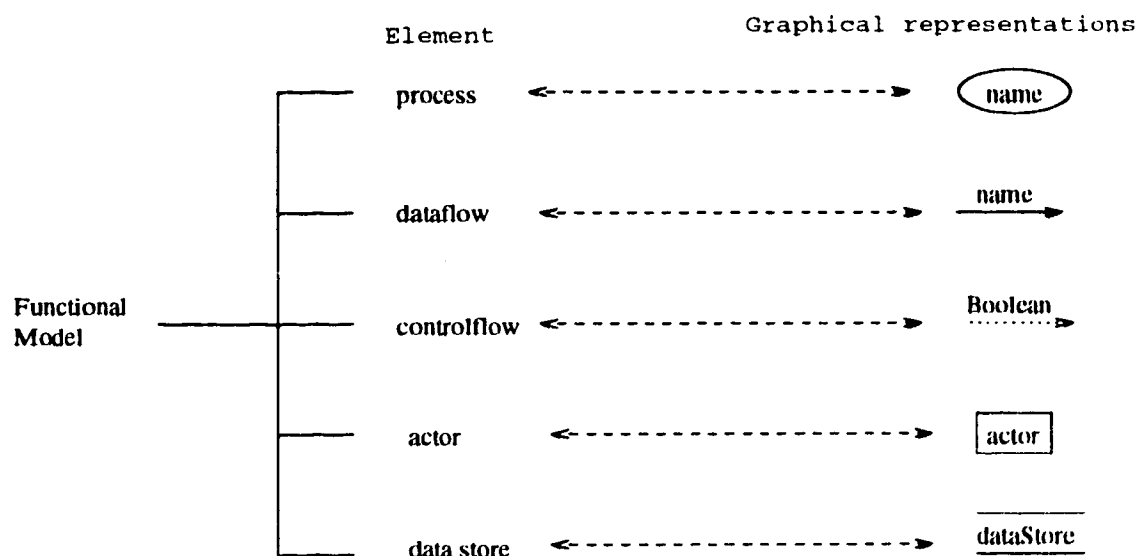


Figure 21: Notations of Functional Model

### Entities:

In the OMT's Functional Model, a **process** is an entity type to transform data, an **actor** is an entity type to produce and consume data, and a **data\_store** is an entity type to store data. The entity type **data\_object** is created to express the data in the OMT support environment. These concepts are modeled as follows:

#### ENTITY\_TYPE

```
process GENERIC IS_A universal_entity;
data_store GENERIC IS_A universal_entity;
actor GENERIC IS_A universal_entity;
data_object GENERIC IS_A universal_entity;
```

### Relationships:

There are three types of flows in a Functional Model: a **data\_flow** to show how a data value is changed by a process or actor, a **fcontrol\_flow** to show how a process can exhibit control over another process, and a **oneValue\_store** to indicate that a process only creates a single value and stores it in a **data\_store**. These relationships are modeled as:

#### RELATIONSHIP\_TYPE

```
data_flow GENERIC IS_A universal_relationship
  ROLES(source, flow, destination)
  PARTICIPANTS (process, data_object, process|actor|data_store)
               (actor|data_store, data_object, process);

fcontrol_flow GENERIC IS_A universal_relationship
  ROLES(source, flow, destination)
  PARTICIPANTS (process, data_object, process);

oneValue_store GENERIC IS_A universal_relationship
  ROLES(source, flow, destination)
  PARTICIPANTS (process, data_object, data_store);
```

### Aggregates:

The **function\_model** is an aggregate with components selected from all elements in the Functional Model. Because OMT supports nested Data Flow Diagrams, the **process** is modeled as a special entity type which can be exploded to the aggregate **function\_model**.

#### AGGREGATE\_TYPE function\_model

```
COMPONENTS (process, data_store, actor, data_object,
            data_flow, fcontrol_flow, oneValue_store
            );
```

#### ENTITY\_TYPE process IS\_A universal\_entity

```
% process can be exploded into a function_model
BECOMES function_model;
```

### 3.4 Defining and Modeling the OOD

OMT does not prescribe a special model for OOD, rather a process of elaborating the OOA model. In this section, we present OOD modeling concentrating on how to apply OMT's OOD guidelines to the prototyped OOA environment.

#### The System Design

A major goal of System Design is to create a system architecture in which the system is decomposed into subsystems with the minimized coupling across subsystems and the maximized cohesion within the subsystems. In Rumbaugh's OOD, a system is decomposed into subsystems using both vertical partitions and horizontal layers. A *subsystem*, which consists of a group of classes, associations, events and constraints, is designed to implement a special service through well-defined interface.

Figure 22 illustrates an example of an application system's decomposition. The system consists of four layers (*computer hardware*, *operating system*, a *middle-ware layer*, and an *application layer*). Each layer makes use of the services of lower-level subsystems and provides the fundamental services to higher-level subsystems. The second *middle-ware* layer subsystem is further decomposed into three vertical partitions to provide three different services. In order to implement its services, the middle partition again is decomposed into three layers of subsystems called *window graphics*, *screen graphics*, and *pixel graphics*.

In OMT environment, a system can be decomposed via the Object Model diagrams. Aggregations are used to implement the abstraction of different levels. That is, a subsystem can be represented as an icon at a higher-level. The icon can be exploded into another object diagram which depicts the subsystem in greater detail. After decomposing a system, the analyst should revise the corresponding Data Flow Diagrams to show explicitly the information flow between the different subsystems.

#### The Object Design

*Object design* is the process of evolving the analysis models into a description of class operations and their implementing algorithms. It may also result in the addition of classes to improve the efficiency of the system. The *object design* has two major

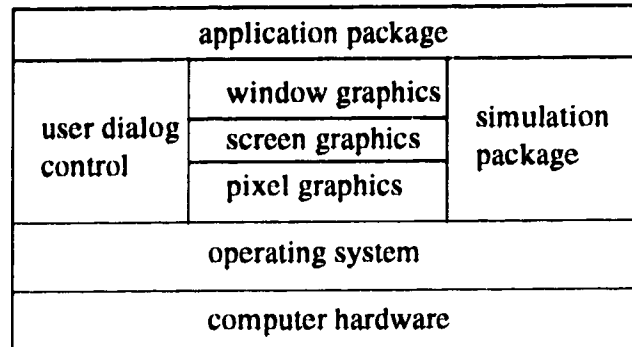


Figure 22: Decomposing system

objectives:

1. *Determine the Full Definitions of Classes and Associations*

First, the signature of how to use an operation and the algorithm of how to implement an operation must be designed. The designer obtains the class, its attributes and its operations from the *object model*. Then, the designer combines the three analysis models to get the implementation of operations for the object model. This is done by converting the actions and activities of the *Dynamic Model* and the processes of the *Functional Model* into operations.

In our OMT prototype, the designer can make use of the *Object Model* developed in the OOA modeling to achieve the Object Design. The text variable *description* is assigned to each class, attribute and operation. In addition, we assign each class a text variable *superclass* to depict the class hierarchy and assign each operation a text variable *algorithm* to depict the implementation of the operation.

2. *Optimizing the System*

The designer optimizes the system by increasing the amount of inheritance, which is implemented using abstract super classes to cover the behavior common to several classes. Shared attributes and operations are moved to abstract superclasses.

### 3.5 Results and Discussion of OMT Modeling

Based on our modeling efforts for OMT in Metaview, we make the following assessments:

1. *Metaview Support O-O Methodology*

Although there are some major differences between the O-O and structural methods in both the notation and modeling strategy, Metaview has the capability to support much of what is required in the OMT environment. Almost all the aspects of OMT method have been modeled in Metaview.

2. *OMT is a Good Method With Strong OOA*

OMT is a complex O-O method and has much in common with other popular O-O methods, such as *Coad and Yourdon* [CY91] and *Shlaer-Mellor* [SM91]. Although its OOD is comparatively weak, OMT introduces a formal method to do object modeling in the OOA. The various notation assure the capability to describe the real-world clearly.

The further assessments about Metaview's modeling capability are:

1. *Metaview's Strengths:*

It is easy to generate an O-O environment using Metaview<sup>2</sup>, because Metaview provides facilities to define an environment. In addition to the conceptual and graphical modeling support, Metaview also supports repository management and graphical editing. The following Metaview's capabilities are particularly helpful in defining an O-O support environment:

- *Aggregation*

Metaview's aggregation is powerful in the sense that it is used to achieve three goals. First, when defining an O-O specification environment, every O-O model of the environment is constructed using aggregation, which consists of all the entity and relationship types of that model. An entity within a model can also have a strong link to another model by aggregating the entity to the linked model. For example, each *class* in the *Object Model* may have a *Dynamic Model*

---

<sup>2</sup>In chapter 5, statistical tables are given to compare the complexity and workload of defining OMT environment and OOM environment in Metaview.

to depict the dynamic behavior of the class. The strong link is created by aggregating the entity type *class* to the aggregate type *Dynamic Model*. The third goal of using aggregation is to produce the nested diagrams. For example, in OMT, a rather complex *intermediate state* (say A) within the *Dynamic Model* may be represented using a nested diagram to analyze A in detail. The nested diagram should inherit all the inputs and outputs of A (e.g. the events and actions), and consist of the entity and relationship types defined for the *Dynamic Model*. When a nested diagram is created, all the related inputs and outputs are automatically included in the nested diagram by Metaview.

- *Generalization*

Generalization is widely used to increase the efficiency of OMT's modeling. For example, the relationship type **many\_to\_many\_associate** is modeled as another relationship type **associate**'s subtype using Metaview's generalization. Many definitions for **many\_to\_many\_associate**, such as its attributes, roles and participations, are inherited from the supertype **associate**.

- *Constraints*

Metaview has strong support for consistency and completeness checking. Dozens of such constraints are defined for the conceptual and graphical modeling perspectives as illustrated in the Appendix.

## 2. Metaview's Limitations

Metaview provides good facilities for modeling the conceptual aspects of an O-O method; however, it is sometimes difficult to build a user-friendly interface for developer. The following limitations exist:

- *Length of the Text*

The developer should not be limited in the length of text. In the current Metaview prototype, if a developer creates very long text string, the text may exceed the maximum length of the display field defined by environment definer and only part of the text appears.

- *Text Fonts*

Different font sizes and styles are necessary for a good CASE environment. At present, Metaview only provides one font size and style.

- *The Scaling Function*

The current Metaview prototype does not support a scaling function. When a graphical icon is selected, it cannot be enlarged or reduced in size. This leads to the problem of dynamically defining the size of an icon. For example, the size of the class icon is difficult to define because a class may have an arbitrary number of attributes and operations drawn within the class box. To solve this problem, in the OMT prototype, class attributes and operations are moved out of the class box so that various attributes and operations can be expressed.

- *The Grouping Function*

Grouping is important for graphical editing. After grouping several graphical representations together, it should be possible to *move*, *copy*, or *delete* the group of graphical objects. It is necessary for Metaview to provide a grouping function to enhance graphical editing and system decomposing.



## Chapter 4

### O-O Methods Review

Recently much work has been undertaken to compare popular object-oriented methods [dCF92, Goo92, YP93, HS91, Loy90]. However, on the basis of comparisons we plan to propose an enhanced method and develop its prototype in a metasystem. The following four methods and corresponding commercial CASE tools are reviewed for this purpose.

1. *Rumbaugh's OMT and OMTool CASE tool*
2. *Booch's method and Rational Rose CASE tool*
3. *The Fusion method* by Hewlett Packard
4. *Shlacr-Mellor's method and Teamwork's OOA/OOD CASE tool*

There are some other popular methods, but their major strengths have generally been subsumed in the above methods. For example, Coad and Yourdon's OOA/OOD method is very similar to OMT [Goo92]; Class Responsibility Collaborator (CRC), which is the major feature of Wirfs-Brock's method [Goo92], has been included in Fusion Method. Another reason of choosing these four methods is their popularity. They are regarded as the representative methods and a number of commercial CASE tools have been developed for them. The experience of running these CASE tools is helpful in comparing the corresponding methods.

Because one of our objectives is to propose an enhanced O-O method, only the strengths of each method are described in detail. During the review of these four methods, the implementation phase and its transformation from OOD are omitted

because the major focus of this thesis is on the object-oriented analysis and design phases.

## 4.1 Rumbaugh's OMT and OMTool

Chapter 3 provides many of the details of the OMT method; therefore, only a summary and critique is presented here.

OMTool was developed by the Advanced Concepts Center (ACC) of General Electric to support Rumbaugh's OMT [Rum91]. The *Object Model*, *Dynamic Model* and *Functional Model* are adopted in the analysis phase of OMT. These three models are used to depict the static aspects of objects and relationships, the lifetime state transition of essential objects, and the computing and processing of the object data, respectively. OMT's OOD includes the *System Design* and *Object Design* phase. The *System Design* aims at configuring the system architecture. A complicated system is divided by vertical *partitions* and horizontal *layers*. The *Object Design* focuses on the object implementation aspect with special emphasis on inheritance, which is considered to increase the system efficiency. For each class, its external interface is adjusted, and its internal data structures and algorithms are determined for every operation defined in the analysis phase.

On the basis of modeling OMT in Metaview and comparing OMT with other methods, it is observed that:

- Overall, OMT is an effective method to develop a wide range of application systems. However, in the analysis phase, OMT lacks system partitioning to handle a very large system. Using partitions and layers to divide classes in the design phase is not enough [Gil94].
- The *Object Model*, *Dynamic Model* and *Functional Model* are very expressive; however, the link between the *Dynamic Model* and *Functional Model* should be strengthened.
- No formal design model and interaction description are provided in OMT.

## 4.2 Booch's Method and Rational Rose

*Rational Rose* is the CASE tool developed by Rational Corporation for Booch's method. The summary of Booch's method is based on our experience with using Rational Rose and reviewing Booch's most recent method description [Boo94].

### 4.2.1 Models in Booch's Method

As shown in Figure 23 [Boo94], Booch uses two dimensions, logical/physical views and static/dynamic views, to capture the structure and behavior of a system during object-oriented analysis and design. The logical view is used to describe the problem domain, while physical view describes the concrete software and hardware composition that is to be implemented. A number of diagrams are denoted for each view model. For the static model, the *Class Diagrams* are used in analysis phase, and the *Module Diagrams* and *Process Diagrams* are used in design phase. *Object Diagrams* are used for both analysis and logical design. To express the dynamic behavior, the *State Transition Diagrams* and *Interaction Diagrams* are introduced.

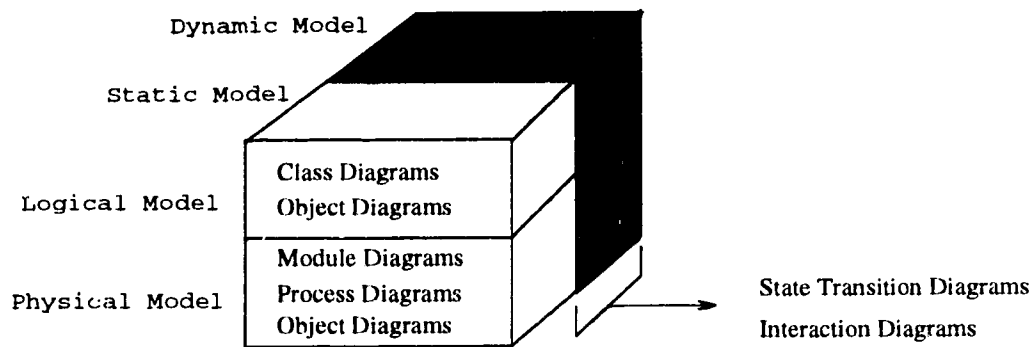


Figure 23: Booch's Method

#### 1. *Class Diagrams*

The graphical notation for a *Class Diagram* is simple and can be regarded as a subset of the notation in OMT's *Object Model*. An exception to this is that *class categories* are used to partition the logical model of a system in a *Class Diagram*. Each class icon identifies the category to which it belongs. This helps

to analyze a complex system. In addition, textual specifications for the *Class Diagrams* include:

```

Responsibilities: text
Attributes      : list of attributes
Operations      : list of operations
Constraints     : list of constrains
State machine   : reference to state machine
Export control  : public | implementation

```

The *state machine* variable refers to the state machine model, which depicts the state transformation of the class. *Export control* defines whether a class is a *public class*, which is visible to other classes, or a *implementation class*, which is invisible.

For each operation within a class, the important specifications are:

```

Return class    : refer to class
Arguments      : list of formal arguments
Export control  : public | protected | private | implementation
Preconditions  : text
Semantics      : text
Postconditions : text
Exceptions     : list of exceptions

```

*Export control* uses such concepts as *public*, *protected*, *private*, and *implementation access* to claim whether the accessibility of an operation is a public, protected, private, or not available from outside. *Preconditions* and *postconditions* are used to describe the operation. The entry of *Exceptions* lists all the exceptions that might happen.

## 2. *State Transition Diagrams*

A *State Transition Diagram* depicts the possible state transitions of an object in its lifetime. It is very similar to OMT's *State Diagram* except that it is less complex. For example, OMT supports concurrency in the *State Diagram* while Booch's Method does not.

## 3. *Object Diagrams*

Objects in the *Object Diagrams* are actually object instances. The diagrams describe two aspects: the objects and relationships, and the second phase, the collaboration between different objects, which includes the objects' visibility to other objects, the communication between objects, and the time constraints (called *time budgets*) for determining when to invoke a message.

#### 4. *Interaction Diagrams*

*Interaction Diagrams* are used to trace the execution of scenarios. In the diagrams, the message passing is extracted from the *Object Diagrams*.

#### 5. *Module Diagrams*

They are used to indicate the physical layering and partitioning of a system. Generally, each module corresponds to a physical file. Modules are connected by arrows to express the dependency. Figure 24 illustrates an example: if module B provides a compiling dependency for module A (e.g. module B is a `.h` file defining classes used in module A), then module A has an arrow directing to module B. The arrow connection between two subsystems shows their dependent relationship.

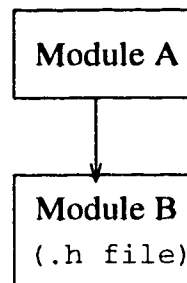


Figure 24: An example of Module Diagram

#### 6. *Process Diagrams*

A *Process Diagram* shows the allocation of processes to processors in the physical design of a system. It is quite similar to the step of *allocating subsystems and tasks to processors* in OMT's *System Design*. Booch gives heuristic rules to achieve this.

### 4.2.2 Summary and Critique

In Booch's method, OOA and OOD are not distinguished clearly, and OOA is rather weakly supported. Its OOD is well supported with the aid of *Object Diagrams*, *Interaction Diagrams* and *Module Diagrams*. As mentioned in Chapter 3, OMT does not have formal models for OOD, and a formal transformation from OOA to OOD is not presented. Booch's Method may address OMT's weaknesses in this regard. Finally, the textual documentation in Booch's Method is helpful for OOA and OOD.

## 4.3 Fusion Method

The Fusion Method [Col93] is a new object-oriented method developed by Derek Coleman *et al* from Hewlett Packard. This method is influenced by several methods including Rumbaugh *et al*'s OMT, Booch's Method, Class Responsibility Collaborator (CRC), and Formal Methods [Pre92] which use a mathematically based specification involving precondition and postcondition statements, to describe the models and avoid ambiguity. Table 4.1 lists all the methods affecting the Fusion Method. The Fusion Method is mainly influenced by OMT in the analysis phase, and Booch's Method in the design phase.

### 4.3.1 OOA

#### OOA Models

Figure 25 presents the structure of Fusion's OOA. The problem domain is described by a *System Object Model* and a *System Interface Model*, both using a data dictionary as a central repository. The System Interface Model is further divided into two models, the *Operation Model* and *Life-Cycle Model*.

Among the OOA models, the *System Object Model* supports the analysis of the objects and relationships within the system boundary. *Life-Cycle Model* is used to describe scenarios of how the system state changes during its life-cycle (from the creation to death). A system and its outside *agents* (such as users or physical devices) communicate through input and output events. A *system operation* is defined as an event and its result. When an input event occurs, a system operation can cause a

FUSION		DERIVING METHODS
O O A	Object Model	OMT's Object Model
	Operation Model	OMT's Functional Model and Formal methods' pre-condition and post-condition specification
	Life-cycle Model	Jackson System Design
O O O D	Object Interaction Graphs	Class Responsibility Collaborator (CRC)
	Visibility Graphs	Booch's Object Diagrams
	Class Description	Booch's Class Diagrams
	Inheritance Graphs	OMT's Object Model

Table 4.1: Relationship Between Fusion and Other O-O Methods

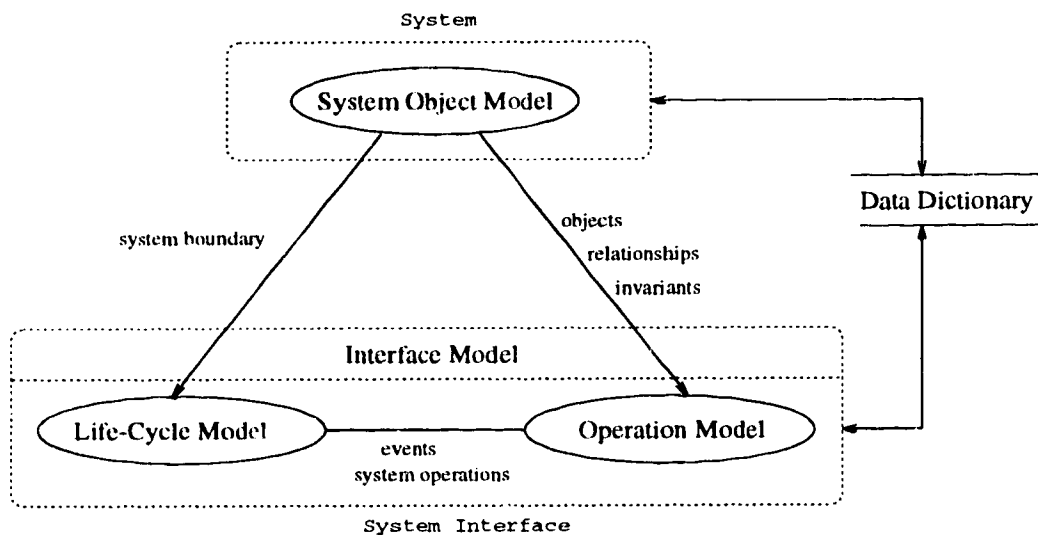


Figure 25: Fusion's OOA

system state change, which may in turn cause an event with its notification sent to outside agents. This idea is quite similar to OMT's *Dynamic Model*. However, the *Life-Cycle Model* is limited to the system level, and does not apply to the object level as does the *Dynamic Model*.

The *Operation Model* captures the details of all the system operations having been analyzed in the *System Object Model* and *Life-Cycle Model*. An *Operation Model* consists of precondition clauses (called *Assumes*), postcondition clauses (called *Results*), and other clauses for recording what an operation does.

Both the *Life-Cycle Model* and *Operation Model* are processed as textual document. A graphical model, such as OMT's *Dynamic Model* and *Functional Model*, is not proposed for these aspects. This weakens the capability for describing how system states and class states change in the Fusion Method.

### OOA Model Integrity

As shown in Figure 25, close relationships exist between the *System Object Model* and *System Interface Models*. In a *System Object Model*, objects, relationships, attributes, and constraints such as *invariants* (assertions on class attributes), are used to analyze the *Operation Model*. The boundary of the *System Object Model* determines the scope of *Life-Cycle Model*.

### Consistency and Completeness Checking in OOA

A remarkable advantage of the Fusion Method is its provision for formal consistency and completeness checking. In Fusion's OOA, three steps, *completeness*, *simple consistency*, and *semantic consistency checking*, are carried out.

#### 1. *Completeness against the requirement*

Iterate the *Analysis Models* until it can be assured that *Life-Cycle Model* covers all the possible scenarios, the *Operation Model* covers all the system operations, and the *System Object Model* captures all the static information from the problem domain.

#### 2. *Simple Consistency*

- (a) Because *Life-Cycle Model* describes the system state changes caused by



external events, the boundary of the *System Object Model* should be consistent with the *Life-Cycle Model*.

- (b) The *Life-Cycle Model* assists in the construction of the *Operation Model*. All the system operations captured in the *Life-Cycle Model* should be included in the *Operation Model*. The *objects, relationships and their invariants* depicted in the *System Object Model* also help to build the *Operation Model*.
- (c) The *Data Dictionary* stores all the identifiers in the analysis models.

### 3. *Semantic Consistency*

- (a) There is a relationship between the *Life-Cycle Model* and the *Operation Model*. The output events of *Life-Cycle Model* should correspond one to one with the results of the system operations depicted in *Operation Model*.
  - (b) Classes, their attributes and relationships in the *System Object Model* should be reflected in the *Operation Model*. In addition, invariants in the *System Object Model* must be logically consistent with the operation precondition and postcondition in the *Operation Model*.
4. As a final step, the developer is recommended to choose several example scenarios as a mechanism for checking semantic consistency.

## 4.3.2 OOD

There are four models in Fusion Method's OOD: the *Object Interaction Graph*, *Visibility Graph*, *Class Description*, and *Inheritance Graph*. Because the first two are very useful for our enhanced method, they are described in detail.

### Object Interaction Graphs

*Object Interaction Graphs* are used to develop system operations described in the *Operation Model*. Each system operation should correspond to an *Object Interaction Graph*, which describes how a system operation is implemented through object interactions and message passing.

Figure 26 shows the graphical representations of an object instance and a collection of objects. An object instance is expressed by a solid box containing the object name. A collection of objects is expressed by a dashed box containing its collection name. Following an object or collection name, a class name is given to identify in which class the object or collection belongs.

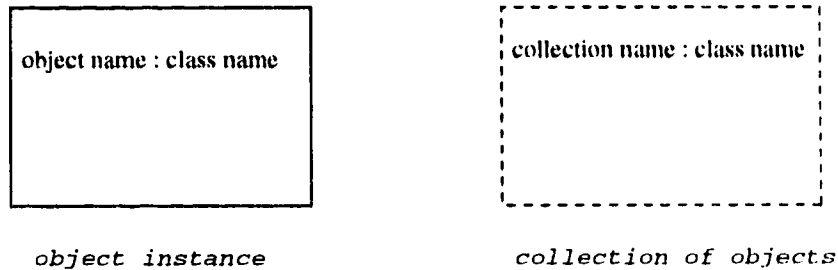


Figure 26: Object instance and collection

In an *Object Interaction Graph*, a message sender is related to its receiver by drawing an arrow directed to the receiver. The name, parameters, and return values of a message are attached as labels on the arrow. The parameters and return values are optional. The graph indicates that when a receiver accepts a message from a sender, it invokes the corresponding method defined in its interface. In the Fusion Method, if a message is sent to a collection, it means all the object instances of this collection receive this message. It may be the case that a message should be passed to only a subset of collection. If this is the case, a constraint condition is placed below the arrow to show this subset constraint. Figure 27 gives an example: *object1* sends *message1* to part of the collection of objects in *collection1*, subject to the constraint of *[subset constraint condition]*.

*Message sequencing* is necessary when a sender sends several messages to the receiver. In this case, the order of the messages is specified using sequence number placed over the message. For example, in Figure 27, *object1* sends *message1* to *collection1* after it receives *message0*. *Message2* is sent when *collection1* executes a corresponding method in response to *message1*. As a result of receiving *message2* from *collection1*, *object1* invokes another message named *message3* and sends it to *collection1*. This sequence can be determined unambiguously based on the sequence numbers (1, 1.1, 2).

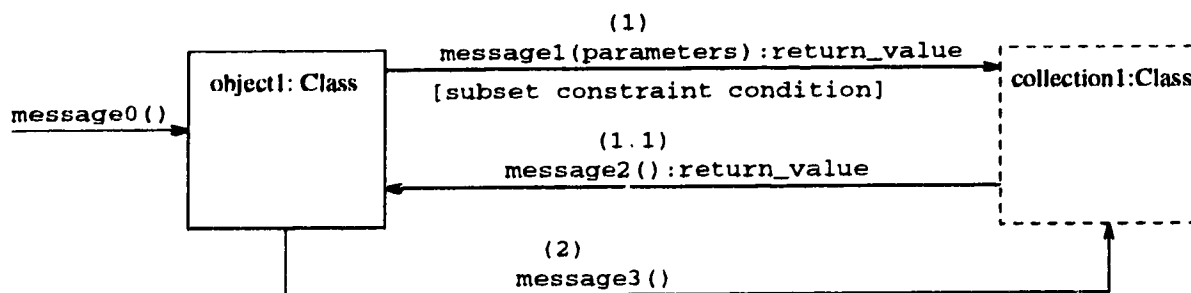


Figure 27: Message sequencing

## Visibility Graphs

Clients and servers are depicted in *Visibility Graphs*. A client has a *reference* (or access) to its server. *Visibility Graphs* illustrate the reference (visibility) structure of classes in the system. They show how the communications depicted in *Object Interaction Graphs* are carried out. In *Object Interaction Graphs*, all objects are assumed to be mutually visible. However, a server object is considered as an exclusive object and can only be accessed by its client(s). The visibility of objects are described using the following characteristics:

- *Reference Lifetime*

When a client needs a shared server throughout its lifetime, a solid arrow is drawn from client to server, otherwise, a dashed arrow is drawn to express that server is only effective for a period. In Figure 28 *server2* is defined as a dynamic server to the *client class*, while *server1* is defined as a permanent server.

- *Server Visibility*

If a server is exclusively used by a client, a double-line border is used to express this characteristic of the server object (e.g. *server3* in Figure 28).

- *Server Binding*

A server can be exclusively used by a client class during the client's lifetime. If such a client is deleted, its bound server is useless and should be deleted also. This property is called *server binding*. In order to show this kind of close coupling, a server box is moved into the boundary of client box. Figure 29 shows the *server binding* between client *Person* and its server object *child*.

- *Reference Mutability*

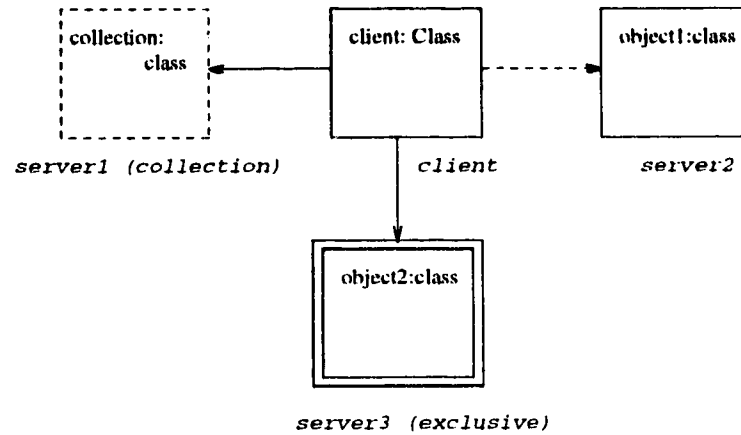


Figure 28: Visibility

*Reference mutability* indicates whether a server can be changed. If it can not, the key word *constant* should be put before the object name, otherwise, the key word *variable* can optionally be used. In Figure 29, the value of the server object *child* can not be changed in its lifetime, and thus its mutability is declared to be constant. Visibility Graphs also show *dynamic reference*. For example, in Figure 29, client class *Person* has a server object *myJob*. This server object is denoted with the prefix *new* before *myJob: Job*. This defines *myJob* as a dynamically created instance of the class *Job*.

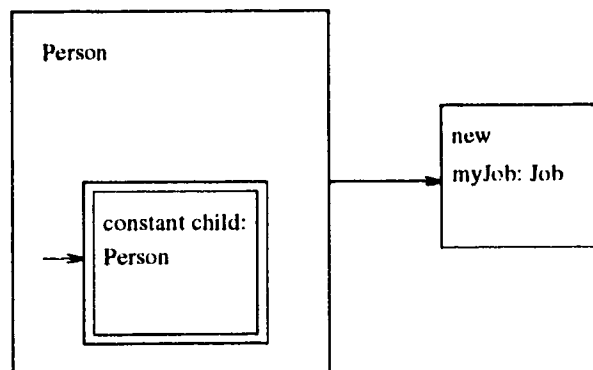


Figure 29: Server binding and reference mutability

The Fusion method has formal consistency checking. *Visibility Graphs* are derived from *Object Interaction Graphs*, and the mapping from latter to former is checked for consistency. Any communications in *Object Interaction Graphs* should be mapped

into a visibility reference in *Visibility Graphs*. In particular, a sender object in the *Object Interaction Graph* must be transformed to a sender class in a *Visibility Graph*.

### Class Descriptions

*Class Descriptions* store all the information about classes, including class inheritance, attribute, and method. The *Class Description* is quite similar to the *textual specification* in Booch's *Class Diagram*. The class inheritance determines whether a class has a superclass, while class attributes and methods form the class interface. In the Fusion Method, each class attribute has its own characteristics including *mutability*, *sharing*, and *binding*. Mutability determines whether a class attribute is a *constant* or *variable*, while sharing indicates whether it is a *shared* or *exclusive* class attribute, and binding determines whether a class attribute is *bound* into a class or not. Class methods contain the operation descriptions (or algorithms) within a class.

Methods are derived from the communicating messages in *Object Interaction Graphs*, while class attributes are derived from the relationships in the *System Object Model*. The class inheritance information can be acquired from *Inheritance Graphs*.

### Inheritance Graph

The fourth component in OOD, the *Inheritance Graph*, uses the same notation as that in OMT's *Object Model* to declare all inheritances. The inheritance relationships in the *System Object Model* and the abstract classes in *Object Interaction Graphs* and *Visibility Graphs* are used to build class inheritance.

#### 4.3.3 Summary and Critique

There are two major contributions among the features of the Fusion Method: the formal design models and the smooth transition from OOA to OOD. Figure 30 [Col93] outlines how the Fusion Method is used in the software development. Regarding requirements as input, the analyst processes the *System Object Model* and *Interface Model*. The *System Object Model* defines the system boundaries, while the *Interface Model* is composed of the *Operation Model* and *Life-Cycle Model*, which depict the system operations and how these operations change the system states in the life-cycle. In the *Life-Cycle Model*, the Fusion Method combines OMT's strategy of

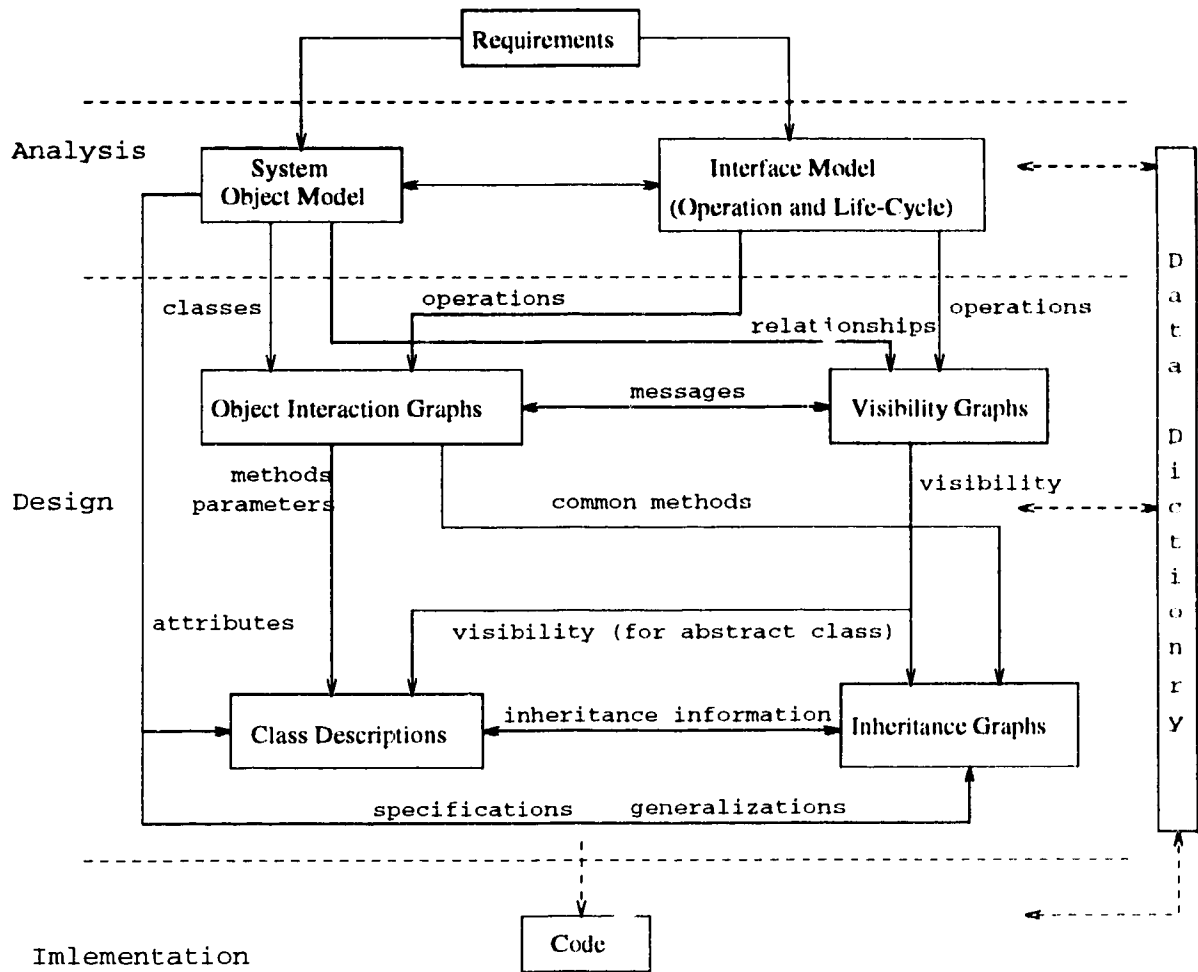


Figure 30: The Fusion Method

finding system operations in the *Functional Model* and a formal method's approach to get precondition and postcondition statements.

Among the four OOD components of the Fusion Method, *Object Interaction Graphs* and *Inheritance Graphs* are used to describe the communication of the objects and how to improve the efficiency of the communication. *Visibility Graphs* and *Class Descriptions* are concerned more with the transformation from OOD to implementation. The *Visibility Graphs* depict the internal states and external interfaces of classes and objects. More detail about the attributes, methods, and inheritances is put into the textual presentation in the *Class Descriptions*.

Although Fusion Method is a rather complete method, the capability of subsystem partition should be strengthened. In addition, in the Fusion Method's OOA phase, concern is on the system operations, while in OMT's OOA, focus is on both system operations (in the *Functional Model*) and object operations (in the *Dynamic Model*). It is necessary to do system analysis before entering OOD phase; however, the disregard of the operations on the object level may bring future problems in the OOD phase for the Fusion Method. The Fusion Method neglects a *Dynamic Model* because of the high complexity of the machine finite state diagrams. But if the analyst does not do any object-level analysis, the work in OOD phase will be much more complex.

## 4.4 Shlaer-Mellor's Method and Teamwork

Shlaer-Mellor's method has some important aspects that support the development of large system. This method is summarized in following subsections based primarily on the work described in [SM91, SM93] and the commercial CASE tool *Teamwork* [Inc93a, Inc93b],

### 4.4.1 OOA

Figure 31 illustrates how OOA is processed in Shlaer-Mellor's method. The concept of *domain* is used to group objects of a large software system by subject areas. The *Domain Chart* depicts how objects in a system are divided into four layers (called domains) according to subject matter and service type. The four domains, from the high level to the low level, are:

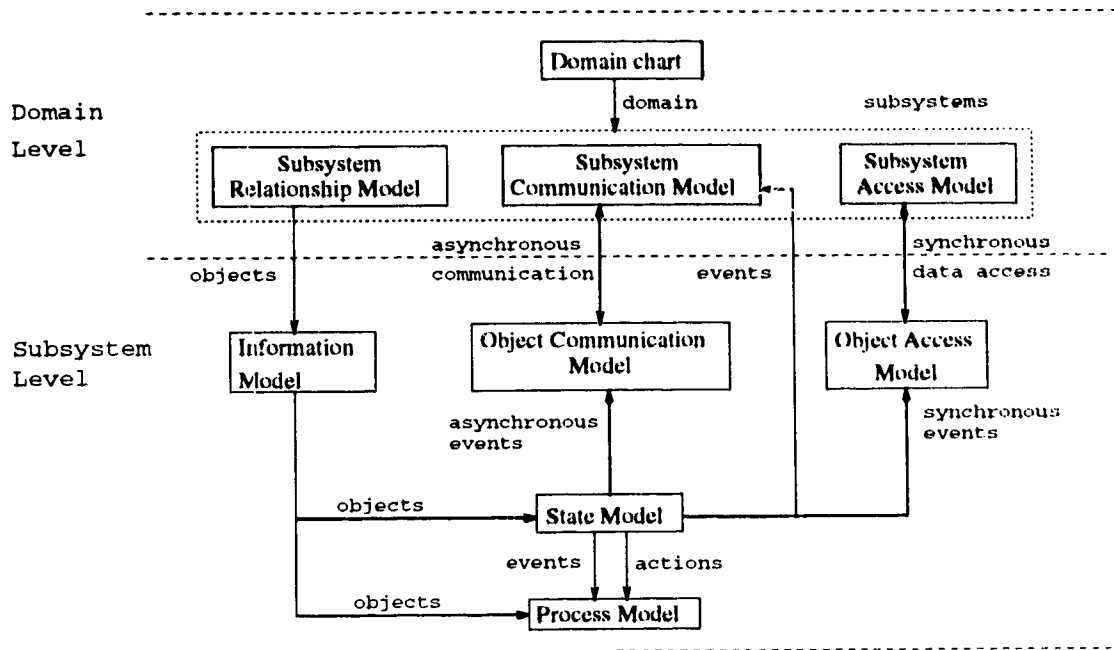


Figure 31: Shlaer-Mellor's OOA

- *Application Domain:* deals with application aspects (e.g., banking operations). There is usually only one application domain.
- *Service Domain:* provides generic services for application domain (e.g., user interface).
- *Architecture Domain:* provides operations that manage the data, and control the system as a whole (e.g., pipes-and-filters architecture).
- *Implementation Domain:* provides the basic operations (e.g., operating system and programming language level facilities).

Although a complex system is split into several levels in the *Domain Chart*, a complex domain can be further partitioned into various subsystems. The relationships between different subsystems are captured by the *Subsystem Relationship Model*. The dynamic aspects of the subsystems should also be depicted. Assume a subsystem (say *X*) has an action (say *A*) which needs to access the data of another subsystem (say *Y*). This is typically accomplished by passing a message from *X* to the subsystem *Y*. If the data access in *Y* is executed when the action *A* has not finished, it is said that a *synchronous communication* happens. The *Subsystem Access Model* provides



the synchronous communication between different subsystems. *Asynchronous communication*, which is message passing without synchronization between subsystems, is depicted by another model called *Subsystem Communication Model*.

After completing the partitioning of a system into domains and subsystems, each subsystem of *Application Domains* and top-level *Service Domains* is further analyzed by using three major models (*Information Model*, *State Model* and *Process Model*) and two subsidiary models (*Object Communication Model* and *Object Access Model*). The objects and relationships in a subsystem are captured by a *Information Model*. The *State Model* uses the *State Transition Diagrams* to analyze the life-cycle of every object which has dynamic behavior within the *Information Model*. The *Process Model* uses *Action Data Flow Diagrams* to expose the details of actions in every state of the *State Model*. These actions are broken down into a number of fundamental processes. Shlaer-Mellor's *Process Model* is different from OMT's *Functional Model*. The *Process Model* captures the actions within each state of a state model, while OMT's *Functional Model* represents the actions among the system. Therefore, Data Flow Diagrams within the *Process Model* are more locally limited in scope.

Synchronous and asynchronous event communications between objects, like those between subsystems, are also depicted by two models: the *Object Communication Model* and *Object Access Model* respectively.

### OOA Model Integrity

Figure 31 also shows the relationships existing between different analysis models:

1. *Domain Chart* provides the complex domain to be partitioned by subsystem models.
2. The subsystem relationships and boundaries depicted in the *Subsystem Relationship Model* help the analyst find all the objects and relationships within a subsystem and then build the *Information Model*.
3. The life-cycle states for each active object in the *Information Model* are depicted in the *State Model*, and the actions in the *State Model* are partitioned into fundamental processes in the *Process Model*. The data processed in the process model is defined in the *Information Model*. The processes can generate events which also appear on the *State Model*.

4. The events between different states within a *State Model* help to define the asynchronous and synchronous communications, which are used to build the *Object Communication Model* and *Object Access Model*, respectively.
5. The *Subsystem Access Model* illustrates the synchronous communication between subsystems and the *Subsystem Communication Model* depicts the asynchronous communication between subsystems. The *Object Communication Model* and the *Subsystem Communication Model* are respectively related to the *Object Access Model* and the *Subsystem Access Model* in a similar manner.

#### 4.4.2 OOD

The OOD phase achieves two tasks: the design of the system into a uniform architecture and the transformation of the OOA models into design models.

The design architecture is shown in Figure 32. The design of a system is composed of a main program, four architecture classes (called *Transaction*, *Finite State Model*, *Active Instance*, and *Timer*) and a number of *Application* classes. The main program is used to initialize classes and operations, trigger the activity of initialized classes by generating the external events of the system, and control the activity of some object instances by calling *Timer*. The *Timer* is used to generate an event at sometime in the future, such as delete a specified instance. The other three architecture classes are used to implement state machine mechanism. Each instance of the *Transition Class* is derived from a transition depicted in a *State Model*, while a *Finite State Model* assembles all the transitions of a *State Model*. An *Active Instance Class* is an abstract class to provide shared state to the subclasses called *Application Classes*.

A state machine must be included as a front-end for each application class. As shown in Figure 33, when an application class receives an event, it invokes the Active Instance class by passing notification of the event. The latter in turn invokes Finite State Model to implement the event by affecting the appropriate state transition. Because a Finite State Model is the assembly of all transition instances of an object, it requires a corresponding Transition class to execute the transition.

The application classes correspond to the classes in the *Application Domain* and the top-level *Service Domain*. After the construction of design architecture, applica-

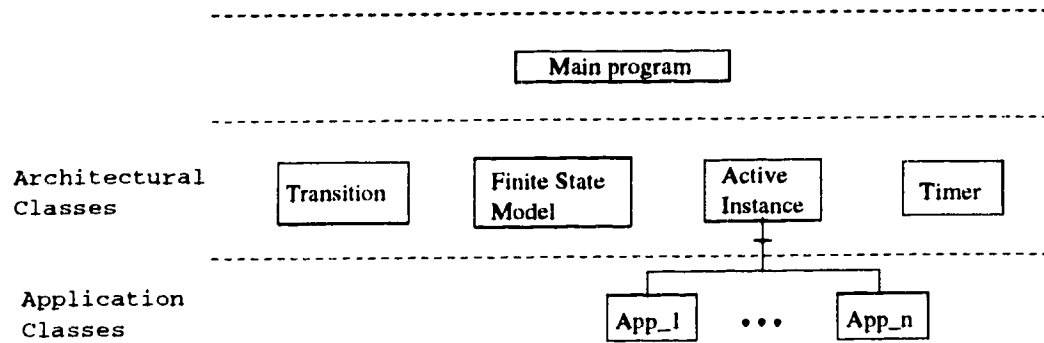


Figure 32: Shlaer-Mellor's OOD architecture

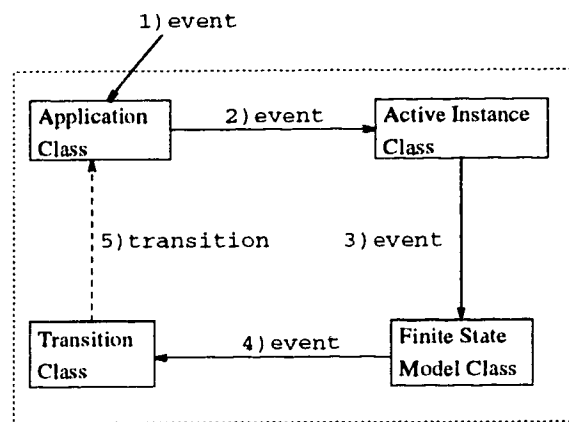


Figure 33: Process of an application class's state transition

tion classes are designed by transferring OOA models into four design components:

- *Dependency Diagram*: It depicts the dependency relationships (client/server or friend) between classes. If a class (say A) accesses the data or operations of another class (say B), class A is said to be the friend of class B.
- *Inheritance Diagram*: It depicts the inheritance relationship between classes. The inheritances between application classes and between the *Active Instance Class* and application classes are captured. In addition, if an application class needs to share data or operations from a class in a *Service Domain*, an inheritance relationship is also built.
- *Class Diagram*: It provides the class interface (or external view).
- *Class Structure Chart*: It provides the class implementation (or internal structure), while taking into the consideration of the operating system and implementation language.

In Shlaer-Mellor's OOD, bridges are built across domains to make the operations of classes in higher-level domains carried out by classes in lower-level domains. For example, the classes in a *Application Domain* can make use of the services provided by the classes in the *Service Domains*. This results high reuse of the classes in the lower-level domains.

#### 4.4.3 Summary and Critique

By providing domain analysis, the Shlaer-Mellor's method is comparatively easy to process a complex system. The domain partitioning and bridging help to build the highly reusable classes in the low-level domains. The transition from the *State Model* to *Process Model* is very straightforward. In Shlaer-Mellor's OOD, the OOA models can be smoothly transformed into OOD models. The uniform architecture design helps generate design products easily. A major disadvantage is that Shlaer-Mellor's method depends heavily on relational database theory. The extended Entity-Relationship model, that is used in the *Information Model*, is a flat model without the *aggregation* relationship.

## **4.5 Conclusion**

1. OMT and Shlaer-Mellor methods are strong in the OOA phase and support the development of large systems. The domain analysis and bridging in Shlaer-Mellor's method provides a particularly straightforward approach.
2. Booch's method is strong in OOD. The textual documents are formal yet very useful as a bridge to the implementation phase. However, Booch's OOA phase is weak. This method is better suited for the development of small to medium size systems in which an extensive analysis phase is not required.
3. The Fusion Method is a rather complete method to build OOA models, and transfer OOA models to OOD models. However, the OOA phase is not strong enough to support large and complex systems.

## **Chapter 5**

# **Enhanced Method—OOM and Its Modeling**

Based on the experience of modeling OMT in chapter 3 and the comparisons of representative O-O methods in chapter 4, an enhanced O-O method called Object-Oriented Modeling (OOM) Method is proposed, defined and prototyped. This method provides a number of complete and consistent OOA and OOD models. The constraints across various models are provided to ensure a smooth transition from OOA to OOD. In addition, OOM is created conscious of the size of the system to be developed. Specifically, some of the OOM models are provided for very large systems development and can be omitted for developing small application systems. This makes OOM both flexible and general when compared to the current popular O-O methods. The defining of the OOM support environment shows that Metaview's modeling capability, especially the capability of defining across-model constraints, is very powerful.

Section 5.1 introduces the OOM Method. Section 5.2 and 5.3 present how the OOA and OOD are defined, respectively. Section 5.4 illustrates how the completeness and consistency constraints are defined in Metaview to support the transformations from OOA to OOD. The chapter concludes with the summary of our experience in using Metaview to define this method.

## 5.1 OOM Method

As shown in Figure 34, there are three phases in OOM: analysis, design and implementation. Since implementation is not a major concern in this thesis, our primary concentration is to build various models in the analysis and design phases.

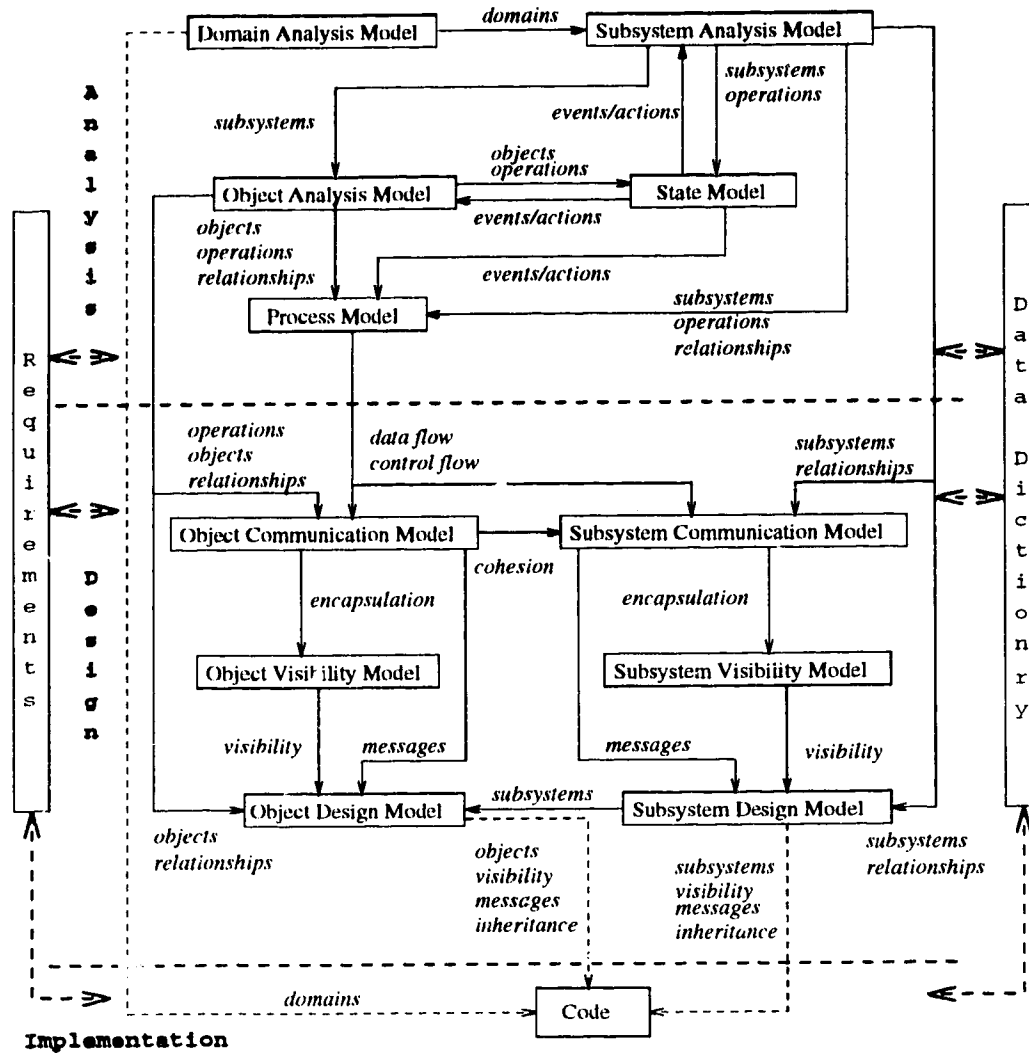


Figure 34: OOM architecture

### 5.1.1 OOA Models

The analysis phase is divided into two steps: the *system-level analysis* which includes the *Domain Analysis Model* and *Subsystem Analysis Model*, and the *subsystem-level analysis* which includes the *Object Analysis Model*, *State Model* and *Process Model*.

A large and complex application system is typically composed of a huge number (greater than 100) of objects and relationships which make the system development difficult. The system-level models are provided to analyze and partition a large system into manageable pieces. After that, the subsystem-level analysis is undertaken for each subsystem depicted within the *Subsystem Analysis Model*. The OOA models are introduced as follows:

#### Domain Analysis Model

A large application system is first analyzed using *Domain Analysis*, which divides the application system into two kinds of subject domains: the *Application Domain* and *Service Domain*. The *Application Domain* is the major layer to deal with application functions, whereas the *Service Domain* provides generic services for application domain, such as user interface or file system management. The client/server relationships between different domains are expressed explicitly in the *Domain Analysis Model*.

The domain partitioning makes it possible for the analysis and design of different domains to be carried out in parallel. In addition, the reusability of subsystems and objects in the *Service Domain* is increased because the generic services can be used in other application systems.

#### Subsystem Analysis Model

A complex domain is partitioned into subsystems. The *Subsystem Analysis Model* is used to describe the subsystems and their relationships within a domain.

#### Object Analysis Model

An *Object Analysis Model* is used to analyze the objects and their relationships in a subsystem. This model is based directly on OMT's *Object Model*; however, some



textual specifications are adopted to each class and object:

**Responsibilities:** text  
**Constraints** : list of constraints  
**Description** : text

The *Constraints* restrict the value range of a class attribute or object attribute. For example, the object attribute *birth-year* could be restricted to be between the year 1850 and current year. Some textual specifications are provided to assist in describing an operation:

**Arguments** : list of formal arguments  
**Export control** : public | protected | private | implementation  
**Preconditions** : text  
**Semantics** : text  
**Postconditions** : text

The *arguments* represent the list of formal arguments. The *export control* defines the accessibility of an operation. And the *semantics* part is used to depict the operation and exceptions that may apply.

## State Model

An active object depicted in the *Object Analysis Model* (or an active subsystem depicted in the *Subsystem Analysis Model*) may carry out a number of object (or subsystem) operations which result in the state transformation. The *State Model* is used to show for each active object or subsystem its possible state sequences throughout its lifetime and the actions that are performed. This model is derived from OMT's *Dynamic Model*. A *State Model* is constructed not only for each object within the *Object Model*, but for each subsystem.

## Process Model

An action of a state depicted in the *State Model* can be analyzed by *Process Model* elaboration to show the fundamental processes for a particular action identified in the *State Model*. This makes the *Process Model* and *State Model* closely connected. Most of the textual specifications for a class operation are adopted for a process specification:

**Arguments** : list of formal arguments  
**Preconditions** : text  
**Semantics** : text  
**Postconditions** : text

### The Advantages of OOM's OOA Models

Two system-level OOA models are the improved models from the Shlaer-Mellor's *Domain Chart* and *Subsystem Relationship Model*. The improvements are:

- Similar to the Shlaer-Mellor's *Domain Chart*, the *Domain Analysis Model* depicts the client/server relation and the services provided by the Server Domain. In addition, the bridges between different domains are explicitly expressed in the *Domain Analysis Model*, which are not available in the *Domain Chart*.
- Although the objective of the *Subsystem Analysis Model* is borrowed from the Shlaer-Mellor's *Subsystem Relationship Model*, the modeling strategy is derived from OMT's *Object Model*. As a result, the expressive capability is stronger than that of the Shlaer-Mellor's *Subsystem Relationship Model*.

The *Object Analysis Model* focuses on capturing the objects and their relationships in the application system. It reflects the static facet of the application system. The *State Model* and *Process Model* are used to capture the dynamic facet. These three models are mainly derived from OMT's analysis models: *Object Model*, *Dynamic Model* and *Functional Model*, respectively. However, some improvements have been made:

- The *State Model* can capture both the system operation's and object operation's life-cycle while OMT's *Dynamic Model* only captures the object operation's life-cycle.
- The *Process Model* can capture both system operations and object operations, whereas OMT's *Functional Model* only captures the system operations.
- OMT's *Dynamic Model* depicts the life-cycle of object operations while its *Functional Model* depicts the system function. The transformation between these two models is not as well defined as that from the OOM's *State Model* to its corresponding *Process Model*. In the OOM Method, a *Process Model* is created for the actions of the states within the *State Model*.

- Textual documentation is provided in OOM's OOA models. For instance, the entity attributes **preconditions** and **postconditions** are used to describe the functions depicted in the *Process Model*.

### 5.1.2 OOD Models

Those O-O methods that are strong in design phase (e.g., Booch's Method and Fusion Method) provide object design models to depict the visibility, inheritance and communications between objects. However, no such description was provided on the system level. This creates problems when very large application systems are developed. In addition to the object design models, the OOM Method also provides the system-level design models, which are processed earlier than the object design for a large application system.

Similar to the Fusion Method, the OOM object design includes the *Object Communication Model*, *Object Visibility Model*, and *Object Design Model*. Because a subsystem can be regarded as a highly abstract object, the visibility, communications and inheritance between subsystems are explored in a similar way as between objects. Therefore, the system-level design includes the *Subsystem Communication Model*, *Subsystem Visibility Model* and *Subsystem Design Model*. The OOM's OOD models are presented as follows:

- **Object Communication Model and Subsystem Communication Model**

The *Object Communication Model* and *Subsystem Communication Model* are used to describe the interactions between different classes and between different subsystems, respectively. The communications show how different objects or subsystems collaborate. The object *operations* and their *relationships* depicted in the *Object Analysis Model*, and the *data flow* and *control flow* depicted in the *Process Model*, are used to help create the *Object Communication Model*. Similarly, the *Subsystem Analysis Model* and *Process Model* are used to help construct the *Subsystem Communication Model*.

The *Object/Subsystem Communication Model* expresses the dynamic interactions (i.e., the message passing) among two or more objects/subsystems.

- **Object Visibility Model and Subsystem Visibility Model**

In an implemented system, a number of server objects provide functions to their clients exclusively. In these cases, the server (or utility) objects are hidden from the other objects. The *Object Visibility Model* is created to describe the visibility of objects. Similarly, the *Subsystem Visibility Model* is constructed to describe the visibility of subsystems.

In the process of building *Visibility Models*, the subsystems and objects depicted in the *Service Domain* are considered to provide services to the subsystems/objects depicted in the *Application Domain*. Therefore, the *Application Domains* and *Service Domains* are connected by the construction of the *Visibility Models*.

### ● Object Design Model and Subsystem Design Model

The *Analysis Model*, *Visibility Model* and *Communication Model* are particularly useful to create a *Design Model*. The communication information helps to define the algorithm details of class operations. The visibility information, which depicts the relationships and visibility of client classes/subsystems and their servers, helps when considering the encapsulation of the invisible classes/subsystems. The inheritance is considered after the two *Design Models* are created on the basis of the *Analysis Models*, *Communication Models*, and *Visibility Models*. Classes having similar properties are reorganized using inheritance and abstract classes. The variable `inheritedfrom` is assigned to each class or subsystem.

Based on the *Object Analysis Model*, *Object Communication Model* and *Object Visibility Model*, and considering the inheritance among objects, the *Object Design Model* is built to provide a complete and detailed object design. Thus, all the objects (including abstract superclass and service objects) and their relationships, as well as the implementation algorithms of each class operation, are captured in the *Object Design Model*.

Similarly, the *Subsystem Analysis Model*, *Subsystem Communication Model* and *Subsystem Visibility Model*, as well as the inheritance of the subsystems (although it is seldom considered), are used to construct the *Subsystem Design Model*.

In conclusion, the static structure and dynamic behavior, as well as the col-

laboration and implementation strategies are captured in the *Object Design Model* and *Subsystem Design Model*, aiming to optimize design and to provide a well-defined transformation from OOD to implementation phase.

### 5.1.3 Implementation Phase

An information system is typically configured as various files in the implementation phase, with each file corresponding to a subsystem. The *Subsystem Design Model* is used to configure the information system, and the *Object Design Model* is needed to help produce files.

As illustrated in Figure 34, both the data repository and the new requirements from users are used throughout the development of the information system. The user requirements provide the feedback from user to system developer and thereby help to ensure the correctness of the system. The data repository stores all the drawings and textual specifications which are used as a basis for code generation the implementation phase. Both the user requirements and repository are helpful for iterating between various models.

### 5.1.4 Transitions Between OOM Models

#### Transitions Between OOA Models

The OOA models are related to each other, as shown in Figure 34:

- Each domain within the *Domain Analysis Model* gives the boundary to its corresponding *Subsystem Analysis Model*, and each subsystem within a *Subsystem Analysis Model* provides the boundary to not only its *Object Analysis Model*, but also the *State* and *Process Models* which depict the system operation behavior.
- The objects and their operations depicted in the *Object Analysis Model* help to build the *State Model* for each (active) object. In a complementary fashion, the *State Model* provides events and actions to refine the *Object Analysis Model*.
- The objects (including their operations and attributes) and their relationships depicted within the *Object Analysis Model* help to define the data flow and control flow within the *Process Model*.

- For each complex action within a *State Model*, a *Process Model* is constructed elaborating on the processes of the action.
- The subsystem is regarded as a higher level abstract object, and therefore there exist similar transformations among the *Subsystem Analysis Model*, *State Model* and *Process Models*.

### Transitions to OOD Models

Transitions to OOD models include not only those from the OOA models, but those from other OOD models. The later designed OOD models may be transformed from the earlier designed OOD models. Such transitions include:

- The *Object Analysis Model*, which depicts the organization of objects together with the *data flow* and *control flow* in the *Process Models*, are particularly helpful in determining the communications between different objects. The *Object Analysis Model* and *Process Model* are used to construct the *Object Communication Model*.
- Similarly, the *Subsystem Analysis Model* and its *Process Models* are used to construct the *Subsystem Communication Model*.
- Some objects in the *Communication Models* merely interact with specific objects and therefore can be regarded as servers exclusively used by their specific clients. With the assistance of the *Communication Models*, the *Visibility Models* are constructed.
- The *Object Design Model* is developed on the basis of the OOA model *Object Analysis Model* and two other OOD models: the *Object Communication Model* and *Object Visibility Model*. The *Object Analysis Model* provides the framework to organize an *Object Design Model*. The messages depicted in the *Object Communication Model* are used to depict the dynamic relationships between different objects, whereas the *Visibility Model* separates the utility objects from the other objects.
- Similarly, the visibility of subsystems depicted by the *Subsystem Visibility Model*, the message communications between various subsystems depicted by the *Subsystem Communication Model*, and the relationships between different subsystems depicted in the *Subsystem Analysis Model*, are used to build the *System Design Model*.

### 5.1.5 The OOM's Flexibility

The OOM Method is developed as a flexible method that can be used in a wide range of application systems. If the application system is a very large and complex system, the *Domain Analysis Model* and *Subsystem Models*<sup>1</sup> are used to divide the application system into manageable pieces. If the application system is a medium size system, the *Domain Analysis Model* is omitted and the *Subsystem Models* are used to partition the application system. If the application system is a small system, the *Domain Analysis Model* and *Subsystem Models* are omitted and only the *Object Models* are used. If the application system is a very small system, some *Object Models*, such as the *Object Visibility Model* and *Object Communication Model*, can also be omitted.

## 5.2 OOA Notation and Its Modeling

Because the subsystem-level models are derived extensively from the OMT support environment, the primary focus in this section is on the system-level models.

### 5.2.1 Subsystem-Level OOA Models

Three subsystem-level OOA models: the *Object Analysis Model*, *State Model* and *Process Model* are derived from OMT's analysis models: the *Object Model*, *Dynamic Model* and *Functional Model*, respectively. Because the defining of OOM's OOA models is heavily reused from the OMT support environment, only the enhancements are discussed.

#### Object Analysis Model and Notation

The enhancements of OMT models to form the Object Model include several textual specifications added to each class, object and operation, and the ability to explode each class to its *State Model*. The *aggregation* function in Metaview is used to

---

<sup>1</sup>Based on the sizes of models, the OOM models are categorized into the *Domain Model*, *Subsystem Model* and *Object Model*. Only the *Domain Analysis Model* is a *Domain Model*, whereas the *Subsystem Analysis Model*, *Subsystem Communication Model*, *Subsystem Visibility Model* and *Subsystem Design Model* are all *Subsystem Models*. The *Object Analysis Model*, *Object Communication Model*, *Object Visibility Model*, and *Object Design Model* are all the *Object Models*.

define the aggregation from a class of the *Object Analysis Model* to its corresponding *State Model* aggregate. The improved notation is illustrated as follows:

#### ENTITY\_TYPE

```
class GENERIC IS_A universal_entity,
  ATTRIBUTES (responsibilities: text, constraints: text)
  BECOMES state_model;
object GENERIC IS_A universal_entity,
  ATTRIBUTES (responsibilities: text, constraints: text)
oo_operation GENERIC IS_A universal_entity,
  ATTRIBUTES (arguments: text, preconditions: text,
    export_control: (public, protected, private, implementation),
    semantics: text, postconditions: text);
```

#### State Model and Notation

Although the *Object Analysis Model* reuses the notation and prototype of OMT's *Dynamic Model*, some improvements are made. The *action* in each state can be optionally aggregated to a *Process Model* which depicts how the action is partitioned into fundamental processes. The *action* entity in the *State Model* is defined as:

#### ENTITY\_TYPE

```
action GENERIC IS_A universal_entity,
  BECOMES process_model;
```

To analyze the system behavior, a *State Model* is created for the *Subsystem Analysis Model*. A variable called *system\_level* is added as an attribute of the *State Model* to distinguish whether the *State Model* is created to analyze the lifetime states of a subsystem or a class.

#### AGGREGATE\_TYPE state\_model

```
ATTRIBUTES (description: text, system_level: boolean)
COMPONENTS (state, transition_name, activity, action,
  state_action, state_activity, control_flow, send_event);
```



## Process Model and Notation

In addition to the notation from OMT's *Functional Model*, the textual specifications for each action are defined in the *Process Model* to provide a more formal description.

### ENTITY\_TYPE

```
process GENERIC IS_A universal_entity,
  ATTRIBUTES (arguments: text, preconditions: text,
              semantics: text, postconditions: text);
```

The *arguments* attribute is used to store a list of formal arguments, and the *semantics* attribute is used to depict the possible processing exceptions.

## 5.2.2 System-Level OOA Models

In the process of defining the *Domain Analysis Model* and *Subsystem Model*, a domain or a subsystem is regarded as a highly abstract object which has attributes and operations.

### Domain Analysis Model

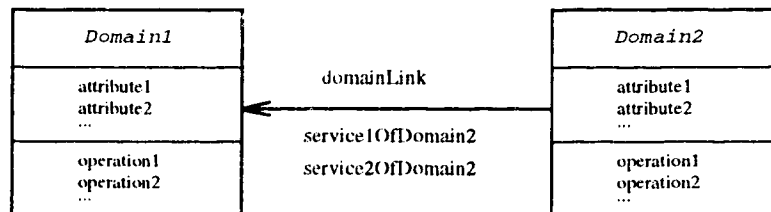


Figure 35: Domain Analysis Model

This model supports the analyst's need of splitting a large application system into several domains. In the process of defining this model, an entity type called **domain** and a relationship type between domains called **domainLink** are modeled. Each domain icon can be exploded into a *Subsystem Analysis Model*. Figure 35 illustrates an example: Two domains, *Domain1* and *Domain2*, are connected by the **domainLink** *domainLink*. *Domain2* provides two services, *service1OfDomain2* and

*service2OfDomain2*, to *Domain1*. The name of each **domain\_link** should be displayed explicitly. It has to be modeled as an entity to participate the relationship **domain\_link**. The entity type **rel\_name** is reused from the *Object Analysis Model* for the name of the relationship **domain\_link**. A easy way to model the services provided by a domain is to model it as an attribute of the relationship type **domain\_link**. However, this makes it difficult to express an arbitrary number of domain services because an entity attribute can store only one domain service. Therefore, the domain service is modeled as an entity type **comment**, which is also reused from the *Object Analysis Model*. A relationship type called **comment\_relate** is created and used to attach a **comment** to its **domain\_link**. Multiple **comment** icons and their corresponding **domain\_link** relationships can show a number of domain services. The *Domain Analysis Model* is defined in EDL as:

#### ENTITY\_TYPE

```
domain GENERIC IS_A universal_entity
    BECOMES subsystem_analysis_model;
```

#### RELATIONSHIP\_TYPE

```
domain_link GENERIC IS_A universal_relationship
    ROLES(domain1, link_name, domain2)
    PARTICIPANTS (domain, rel_name, domain);
```

#### AGGREGATE\_TYPE domain\_analysis\_model

```
COMPONENTS (domain, domain_link,
    % derived from the object_analysis_model
    rel_name, comment, comment_relate,
    oo_attribute, oo_operation,
    entity_attribute, entity_operation);
```

### Subsystem Analysis Model

In the *Subsystem Analysis Model*, entities and relationships are reused from the *Object Analysis Model*, except the newly created entity types **subsystem** and **state\_model\_name** and the relationship type **subsystem\_explode**. A **subsystem** can be optionally aggregated into its static aspects, which is a *Object Analysis Model*. To depict

ing model `object_and_subject` (e.g., which is composed of all the entity and relationship types, as well as conceptual and graphical constraints for this O-O model. The environment defines then identify the specific entity types and their relationship types for each O-O model. The constraints within each O-O model and across different models should be identified next. The final definition of an O-O support environment appears in two files: the `concept` and its graphical counterpart `graphical-con`.

The aggregation and generalization functions provided by Metaview are used to create an efficient representation of an O-O support environment. All the predefined entity and relationship types and constraints are considered for reuse by including them in the model aggregates. For example, in the Object Model, several entity and relationship types and various constraints are reused from the Object Analysis Model. The generalization function is used to define the entity and relationship types which share common properties. Using generalization can increase the code sharing and optimize the defined O-O supporting environment. As examples, the abstract entity `control_center` is used to define its subtypes `forwarder`, `sky_controller`, and `port_controller`. And the abstract relationship `undirected_associate` is defined for two subtypes `undirected_many-to-many` and `undirected_many-to-one`. The participants and roles of the relationship `undirected_many-to-many` and `undirected_many-to-one` are defined in their subtype relationship `undirected_associate`. Thus, any changes for the participants and roles only need to be done for the `undirected_associate`. Moreover, the subtype is the representative for its subtypes. So only the subtype `undirected_associate` is included in its corresponding model aggregate `object_and_subject`. The subtypes `undirected_many-to-many` and `undirected_many-to-one` are not included.

Iteration is necessary for defining an O-O environment. After an O-O model is modeled, the corresponding environment is configured in the Metaview system. The environment defines then test the environment by creating all the defined entities and their relationships, and testing all the defined constraints. Tests that demonstrate both satisfaction and non-satisfaction of each constraint must be developed and then applied. Next some application systems should be specified using the supporting environment. If problems appear, the environment defined can modify the environment quite easily in Metaview. Many iterations may be required to improve the graphical expression of entity and relationship types, and accuracy and completeness of

```

ALL aggri FROM [object_visibility_model]:
*.name == aggri.visibility_model_name]
% s_name is the server_name in the Object Visibility Model.
ALL s_name FROM [server_name@aggri]
% child is the exclusive_servers of s_name.
WHERE child = _child(s_name) SATISFY
% all the exclusive_servers depicted in the Object
% Visibility Model should be included in its Design Model.
SOME s_name FROM [server_name@aggri] SATISFY
_child(name) == child

OTHERWISE
message("Server ", s_name.name, " in object_visibility_model",
aggri.name, " is not correctly duplicated in the
object_design_model.");

```

There are five similar across-model constraints to the subsystems. The difference in definition involves the replacement of the object models by corresponding subsystem models, e.g., using the subsystem model *Subsystem Analysis Model* in place of the object model *Object Analysis Model*.

## 5.5 The Experience of Defining OOM Using Metaview

As prerequisites to modeling an O-O environment using Metaview, it is most important to commit the time and effort to thoroughly understand Metaview's modeling mechanism and the O-O method to be modeled. Approximately 200 hours were spent to understand Metaview and its modeling mechanism. Several existing environments such as Data Flow Diagram (DFD), Higher-Order Software (HOS), Ward-Mellor and TELOS, were studied for understanding how to define a specified environment using Metaview. It took about 150 hours to learn OMT and design how to model OMT. In the implementation phase, approximately 95 and 63 hours were needed for defining the OMT and OOM environments, respectively.

After understanding an O-O method, different O-O models, such as OOM's subsystem Analysis Model and Object Analysis Model, are first identified for the O-O method. Every O-O model (e.g., Object Analysis Model) is defined as a corresponding

5. All the **class** entities depicted in the Object Visibility Model must be included in the Object Analysis Model.

The fourth constraint listed above is a complex constraint, whereas the rest of the constraints are quite similar and are simple ones. Only the definitions of the first and fourth constraints are illustrated:

**Across-Model Constraint 1:** All the classes in a Object Analysis Model should be included in its Object Design Model.

At first an Object Design Model's attribute called *analysis\_model\_name* is added. When the designer assigns the actual name of an Object Analysis Model to its corresponding design model's attribute *analysis\_model\_name*, the link between the Object Design Model and Object Analysis Model is created.

Then the ECL is used to define the constraint across the Object Analysis Model and Object Design Model. In the definition, the digital number 5 represents the fifth constraint provided in Metaview, which is a completeness constraint.

```
CONSTRAINT (5) design_model_completed (aggr2: object_design_model)
% find each Object Design Model's corresponding Object Analysis Model.
ALL aggr1 FROM [object_analysis_model:
    *.name == aggr2.analysis_model_name]
% all the classes depicted in the Object Analysis Model should be
% included in the Object Design Model.
ALL class1 FROM [class@aggr1] SATISFY
SOME class2 FROM [b_class@aggr2] SATISFY
    class1.name == class2.name
OTHERWISE
    message("The class", class1.name, "in object_analysis_model",
aggr1.name, "should also be captured by current design model.");
```

**Across-Model Constraint 4:** All the exclusive servers depicted in the Object Visibility Model should also be included in the corresponding Object Design Model.

```
CONSTRAINT (5) design_model_completed (aggr2: object_design_model)
% find each Object Design Model's corresponding Visibility Model.
```

6. The **server\_name** icon in the Object Visibility Model or Object Design Model can be exploded into an **exclusive\_servers** aggregate, which depicts various exclusive servers of a **class** in terms of single or collection server.
7. The  **subsystem** icon in the Subsystem Design Model can be exploded into an Object Design Model diagram.

Because the actions associated with an aggregation operation are quite similar, only the first aggregation is illustrated:

```
ENTITY_TYPE
domain GENERIC IS_A universal_entity
BECOMES subsystem_analysis_model;
```

### Across-Model Constraints

A number of completeness and consistency constraints are provided by MetaView [Fin94b]. Among them, a completeness constraint (noted as fifth-level constraint) is used to define the constraints across different models, which ensure complete and consistent transformations between different models. For example, all the entities defined in a model (say, modelA) must be included in another model (say, modelB). Before defining a constraint across these two models, a weak link between modelA and modelB must be created. A newly created attribute of modelA is assigned the actual name of modelB, thus a weak link is constructed from modelA to modelB. Then the constraints can be defined along the weak link between two models. The major across-model constraints for the object models are:

1. All the **class** entities in the Object Analysis Model should be included in the Object Design Model.
2. All the **class** entities depicted in the Object Visibility Model should be included in the Object Design Model.
3. All the **server\_name** entities depicted in the Object Visibility Model should be included in the Object Design Model.
4. The entity type **server\_name** may have exclusive servers depicted in its aggregate **exclusive\_servers**. All the **exclusive\_servers** depicted in the Object Visibility Model should be included in the Object Design Model.

These two modeling techniques help to provide well-defined transformations between models. Such transformations are particularly important between OOA models and OOD models.

Aggregation

As shown in Figure 38, an entity in a model can be exploded to another model which defines this entity in detail. Such an aggregation is used to create strong link between an entity and its aggregate.

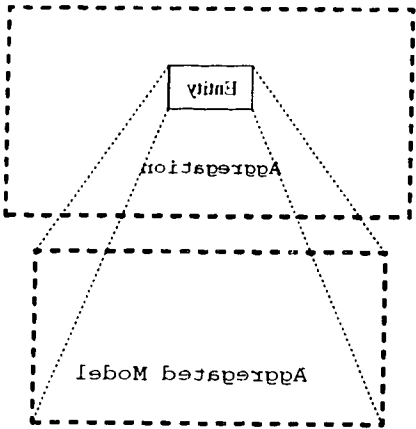


Figure 38: Aggregation to create strong link

There are seven aggregations used in the definition of the OOM support environment:

1. The **domain** icon in the Domain Analysis Model can be exploded into a **Subsystem Analysis Model** diagram by click the **domain** icon.
2. The **subsystem** icon in the Subsystem Analysis Model can be exploded into a **Object Analysis Model** diagram by click the **subsystem** icon.
3. The **state model name** icon in the Subsystem Analysis Model can be exploded into a **State Model**.
4. The **class** icon in the Object Analysis Model can be exploded into a **State Model**.
5. The **state model's action** icon within an **intermediate state** can be further exploded into a **Process Model**.

```

comment_relate, linkAt_relate, linkAt_ternary,
general_constraint,
% entity and relation reused from the Object Visibility
% Model
server_name, servername_link )
ATTRIBUTES (analysis_model_name : text, description : text,
visibility_model_name : text);

```

Only one entity type called `d_object` is defined. All the other entity or relationship type definitions are reused from the Object Analysis Model and Object Visibility Model. Two special attributes called the `analysis_model_name` and `visibility_model_name` are declared for the `object_design_model` to define the consistency and completeness checking that must hold between the Object Analysis Model/Object Visibility Model and the Object Design Model.

To make some relationship types reusable, the entity type `d_class` is added as participant to the corresponding relationship type definitions. For instance, the `servername_link` is defined as:

```

RELATIONSHIP_TYPE
servername_link GENERIC IS_A universal_relationship
ROLES (sender, message, receiver)
PARTICIPANTS (v_class|d_class, server_name);

```

### Subsystem Design Model

The defining of this model is quite similar to the Object Design Model except that in the Subsystem Design Model, the entity type `d_subsystem` is defined in place of `d_class`.

## 5.4 Defining Transformations Across Models

In the process of defining the OOM support environment using Metaview, the `ex-plotting` (or `aggregation`) function provided by Metaview is used to create strong links between different models. In addition, the `across-model` constraints supported by Metaview are used to check completeness and consistency between different models.



the entity `v_subsys` as the participants to the originally defined relationship types. These relationship types are refined as:

```

RELATIONSHIP_TYPE
tmp_link GENERIC IS_A universal_relationship
    ROLES(sender, receiver)
    PARTICIPANTS(v_class, v_object|v_collection)
    (v_subsys, v_subsys);
lifetime_link GENERIC IS_A universal_relationship
    ROLES(sender, receiver)
    PARTICIPANTS(v_class, v_object|v_collection);
    (v_subsys, v_subsys);
server_name GENERIC IS_A universal_relationship
    ROLES(sender, message, receiver)
    PARTICIPANTS(v_class|v_subsys, server_name);

```

### Object Design Model

This model is derived from the Object Analysis Model and Object Visibility Model. This model also focuses on the inheritance of the objects. The `exclusive_servers` aggregate depicted in the Object Visibility Model is reused in the sense that if a class includes a `server_name` in its icon box, the designer can click the `server_name` to open the corresponding `exclusive_servers` aggregate. The Object Design Model is defined as follows:

```

AGGREGATE_TYPE object_design_model
    COMMENTS(d_class,
        % entities reused from the Object Analysis Model
        rel_attribute, oo_attribute, oo_operation, object,
        constraint, qualifier, comment, rel_name,
        control_center,
        % relationships reused from the Object Analysis Model
        link, instant, entity_attribute, entity_operation,
        associate, qualified_associate, ternary_associate,
        aggregate_from, aggregate_to, generalize_from,
        generalize_to, propagating_operation, qualifier_relate,

```

```

AGGREGATE_TYPE object_visibility_model
COMMENTS (v_class, v_object, v_collection,
server_name, tmp_link, lifetime_link,
servername_link);

```

### Subsystem Communication Model and Subsystem Visibility Model

The modeling of the Subsystem Communication Model and Subsystem Visibility Model is quite similar to the modeling of the Object Communication Model and Object Visibility Model respectively.

To define the Subsystem Communication Model, the entity type *subsystem* and the relationship type *subsys\_mes\_passing* are used to replace Object Communication Models *c\_object* and *message\_passing*, respectively. The Object Communication Models entity types *message* and *condition* and the relationship type *condition\_message* are reused. This model's aggregation *subsys\_mes\_communication\_model* is defined as:

```

AGGREGATE_TYPE subsystem_communication_model
COMMENTS (subsystem, messages, condition,
subsys_mes_passing, condition_message);

```

To define the Subsystem Visibility Model, the entity type *v\_subsys* is used in place of the *v\_class* defined in the Object Visibility Model. The other entity and relationship types are reused from the Object Visibility Model and are shown as the components of the aggregation *subsys\_visibility\_model*:

```

AGGREGATE_TYPE subsystem_visibility_model
COMMENTS (v_subsys, v_object, v_collection, server_name,
tmp_link, lifetime_link, servername_link)
ATTRIBUTES (analysis_model_name : text, description : text);

```

The participants to the reused relationship types, *tmp\_link*, *lifetime\_link* and *servername\_link*, must be changed to *v\_subsys* from *v\_class*, because the *v\_class* is defined only for the Object Visibility Model. To define an OOM environment efficiently, the originally defined relationship types are refined to be reusable by adding

## ENTITY\_TYPE

```

v_class GENERIC IS_A universal_entity;
v_object GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text, new_created: Boolean);
v_collection GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text);
server GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text, new_created: Boolean);
server_collection GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text);
server_name GENERIC IS_A universal_entity
BECOMES exclusive_servers;

```

There are two types of relationships between a visible class `v_class` and a visible object `v_object` or visible collection of objects `v_collection`: the temporary link `tmp_link` and the permanent link `lifetime_link`. The `server_name` has a visible link to the `v_class`. When a `v_class` has an exclusive server diagram, the diagram can be exploded by clicking the `server_name` which is displayed within the `v_class` box. The relationships are modeled as:

## RELATIONSHIP\_TYPE

```

tmp_link GENERIC IS_A universal_relationship
ROLES (sender, receiver)
PARTICIPANTS (v_class, v_object|v_collection);
lifetime_link GENERIC IS_A universal_relationship
ROLES (sender, receiver)
PARTICIPANTS (v_class, v_object|v_collection);
servername_link GENERIC IS_A universal_relationship
ROLES (sender, message, receiver)
PARTICIPANTS (v_class, server_name);

```

There are two aggregations: the `exclusive_servers` and `object_visibility_model`, which are modeled as:

## AGGREGATE\_TYPE exclusive\_servers

```

COMPONENTS (server, server_collection);

```

```

message("The condition of a message should be in
a pair of brackets.");

OTHERWISE
properly_enclosed(str)
ALL str FROM [condition @ aggregate] SATISFY
(aggregate: object_communication_model)
CONSTRAINT (5) condition_checking

char == '[' AND last_char_is(']', list);
SATISFY
char = _head(list)
WITH list = _elements(str);
PREDICATE properly_enclosed(VALUE str: string)
```

### Object Visibility Model

Figure 37 illustrates the six major entities of an Object Visibility Model: the `v_class`, `v_object`, `v_collection`, `server`, `server_collection` and `server_name`. A `server_name` can optionally be exploded into an `exclusive_servers` displaying all the exclusive servers which in this case can be a `server` or `server_collection`. The entities are modeled as:

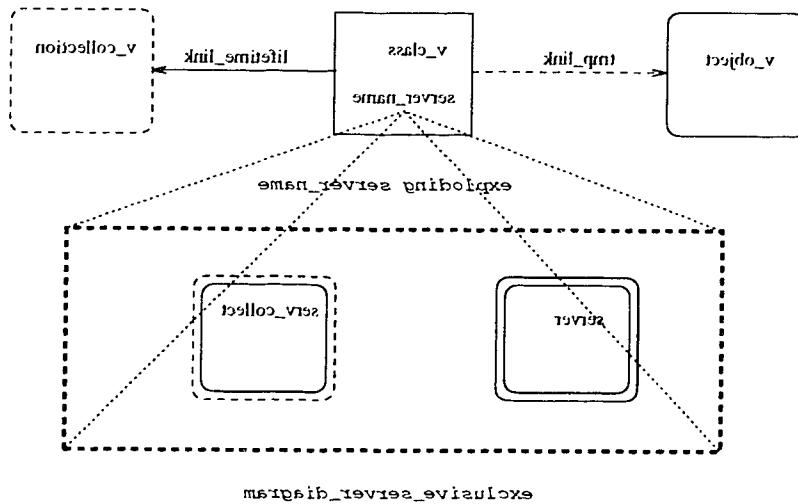


Figure 37: Object Visibility Model

```
ATTRIBUTES (content : text);
```

The relationship type `message_passing` captures how a message is sent from a `c_object` to another `c_object` or to a `collect_object`. `condition_message` uses an invisible link to connect a message and its constraint on the screen. A `c_object` has an arbitrary number of attributes and operations. Similar to the relationship types `entity_attribute` and `entity_operation` for the entity type class in the Object Analysis Model, `cobject_attribute` and `cobject_operation` are defined to depict the relationships of entity `c_object` to its attributes and operations, respectively. The relationship types are modeled as:

```
RELATIONSHIP_TYPE
message_passing GENERIC IS_A universal_relationship
  ROLES(sender, message, receiver)
PARTICIPANTS (c_object, message, c_object|collect_object);
condition_message GENERIC IS_A universal_relationship
  ROLES(condition, message)
PARTICIPANTS (condition_name, message_name);
cobject_attribute GENERIC IS_A universal_relationship
  ROLES(entity_body, attribute_name)
PARTICIPANTS (c_object, oo_attribute);
cobject_operation GENERIC IS_A universal_relationship
  ROLES(entity_body, operation_name)
PARTICIPANTS (c_object, oo_operation);
```

Several constraints are modeled for the Object Communication Model. For example, objects in the Object Communication Model should form a connective graph. A special type of constraint, such as the condition of a message should be in a pair of brackets, could not be defined in previous ECL. To solve this problem, the predicates of checking special character(s) are provided in the new ECL. To check the proper input of `condition`, the predicate `properly_enclosed` is defined to check whether a string is enclosed in a pair of brackets. Some procedures, such as `_always` are ECL system procedures. There are a number of constraint types. The digital number 5 at the beginning of constraint checking indicates that it is a level 5 constraint. It is executed after constraints at levels less than 5 have been applied.

5.3 OOD Notation and Its Modeling

Object Communication Model

The operations, objects and relationships in the Object Model and the data flow and control flow in Process Model are used to help create the Object Communication Model. This model is derived from the Fusion's Object Interaction Graphs, where one object sends a message to a server object when it requires services of another object. The Object Communication Model consists of six entity types (the c\_object, collect\_object, message, condition, oo\_attribute, and oo\_operation) and four relationship types (messagepassing, conditionmessage, collectattribute, and collectoperation). Figure 36 illustrates an example of Object Communication Model. A communication object send a message message1():return\_value1 to the collection of objects collect. The condition [subset constraint condition] is used to constrain the message.

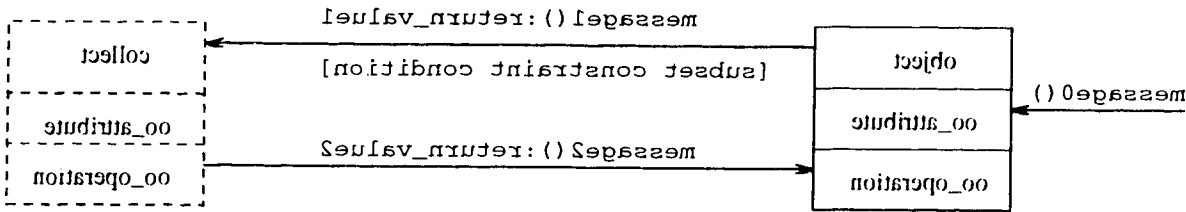


Figure 36: Object Communication Model

Among the entity types, condition depicts the constraints of a message between two objects or a c\_object and a collect\_object. The newly created entities are modeled as:

```
ENTITY_TYPE
c_object GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text);
collect_object GENERIC IS_A universal_entity
ATTRIBUTES (class_name: text);
message GENERIC IS_A universal_entity
ATTRIBUTES (content : text);
condition GENERIC IS_A universal_entity
```

the subsystem's dynamic aspects, a *subsystem* should be able to explode to another aggregate State Model. At present EDL does not support multiple-aggregation, an indirect way is used to link subsystem's State Model. The entity type *state\_model\_name* is used to links *subsystem* and its aggregate State Model via the relationship type *subsystem\_explode*. This model is defined as:

```

ENTITY_TYPE
subsystem GENERIC IS_A universal_entity
BECOMES object_analysis_model;

state_model_name GENERIC IS_A universal_entity
BECOMES state_model;

RELATIONSHIP_TYPE
subsystem_explode GENERIC IS_A universal_relationship
ROLE(subsystem, aggregate_name)
PARTICIPANTS (subsystem, state_model_name);

AGGREGATE_TYPE subsystem_analysis_model
COMPONENTS (subsystem, state_model_name, subsystem_explode,
% entities reused from the Object Analysis Model
rel_attribute, oo_attribute, oo_operation, object,
constraint, dualifier, comment, rel_name,
control_center,
% relationships reused from the Object Analysis Model
link, instant, entity_attribute, entity_operation,
associate, dualified_associate, ternary_associate,
aggregate_from, aggregate_to, generalize_from,
generalize_to, propagating_operation, dualifier_relate,
comment_relate, linkAttr_relate, linkAttr_ternary,
general_constraint);

```

- [Zha93] Yuchen Zhai. Multiple views for integrated case environments. Master's thesis, Department of Computing Science, University of Alberta, 1993.
- [YP93] George Yuen and Nixon Patel. An exploration of object-oriented methodologies for system analysis and design. Proceedings of the workshop on Studies of Software Design pages 184-198, 1993.
- [Ver91] T. F. Verhoef, A. ter Hofstede, and G. M. Wijters. Structuring Modeling Knowledge for CASE Shells. Advanced Information Systems Engineering: Third Intl Conf of CASE'91, pages 503-524, May 1991.
- [The93] The Advanced Concept Center, General Electric. OMTool Demo Version 1.0. 1993.
- [ST93] P.G. Sorenson and J.P. Tremblay. Using a Metasystem Approach to Support and Study the Design Process. Workshop on Studies of Software Design, pages 168-183, May 1993.
- [Sot91] P.G. Sorenson, J.P. Tremblay, and A.J. McAllister. The EARA/GI Model for Software Specification Environments. Technical Report TR 91-14, University of Alberta, June 1991.
- [Sot91] P.G. Sorenson, J.P. Tremblay, and A.J. McAllister. The EARA/GI Model for Software Specification Environments. Technical Report TR 91-14, University of Alberta, June 1991.
- [Smo91] K. Smolander, K. Lyytinen, V. P. Tahvanainen, and P. Martin. Metaphis: A Flexible Graphical Environment for Methodology Modeling. Advanced Information Systems Engineering: Third Intl Conf of CASE'91, pages 168-191, May 1991.
- [SM93] Sally Shlaer and Stephen J. Mellor. A deeper look at the transition from analysis to design. Journal of Object-Oriented Programming pages 16-21, February 1993.
- [SM91] Sally Shlaer and Stephen J. Mellor. Object Lifecycles - Modeling the world in states. Yonendon Press, 1991.
- [Rum94] James Rumbaugh. The Life of an Object Model: How the Object Model Changes During Development. Journal of Object-Oriented Programming, pages 24-32, March 1994.



- [Rum91] James Rumbaugh, et al. Object-Oriented Modeling and Design. Prentice Hall, 1991.
- [Pre92] Roger S. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, Inc., 1992.
- [MG93] Beth Millar and Dinesh Gladwal. EDL System's Manual. Department of Computational Science, University of Saskatchewan, January 1993.
- [McA88] Andrew J. McAllister. Modeling Concepts for Specification Environment. PhD thesis, Department of Computational Science, University of Saskatchewan, 1988.
- [Loy90] P.H. Loy. A Comparison of Object-Oriented and Structured Development Methods. ACM SIGSOFT Software Engineering Notes, pages 44–48, 1990.
- [Lee92] Jesse Ka-Leung Lee. Implementing adiasa transformations in the metawiew metasytem. Master's thesis, Department of Computing Science, University of Alberta, 1992.
- [KM90] Tim Koton and J. D. McGregor. Understanding Object-Oriented: a Unified Paradigm. Communications Of The ACM, pages 40–60, September 1990.
- [Inc93b] Cadre Technologies Inc. Teamwork\OOD User's Guide. 322 Richmond St. Providence, RI 02903-9990, release 2.0 edition, 1993.
- [Inc93a] Cadre Technologies Inc. Teamwork\OOA User's Guide. 322 Richmond St. Providence, RI 02903-9990, release 2.0 edition, 1993.
- [H291] Brian Henderson-Sellers. A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation: a New Approach to Software Engineering. Prentice Hall, 1991.
- [Goo92] Gert Gloor, Shinguan Hong, and Jiaak Brinkkemper. A Comparison of Six Object-Oriented Analysis and Design Methods. University of Twente, 1992.
- [Gim93] Dinesh Gladwal and Beth Millar. EDL User's Manual. Department of Computational Science, University of Saskatchewan, January 1993.

- [Gill94] C. Gilliam. An Approach for Using OMT in the Development of Large Systems. *Journal of Object-Oriented Programming* pages 50-59, February 1994.
- [Gad94b] Dinesh Gadwal, Pius Lo, and Beth Millar. \GIE Users Manual. Department of Computing Science, University of Alberta, and Department of Computational Science, University of Saskatchewan, May 1994.
- [Gad94a] D. Gadwal, P. S. Findeisen, P. G. Sorenson, J. P. Tremblay, and L. B. Millar. Generating Real-time Systems Development Environment Using Metaview. Technical Report TR 94-2, Department of Computational Science, University of Saskatchewan, 1994.
- [Fin94d] Piotr Findeisen. The EARA Model for Metaview. Department of Computing Science, University of Alberta, June 1994.
- [Fin94c] Piotr Findeisen. (Re)Designing ECL. Department of Computing Science, University of Alberta, March 1994.
- [Fin94b] Piotr Findeisen. Environment Constraint Language for Metaview. Department of Computing Science, University of Alberta, May 1994.
- [Fin94a] Piotr Findeisen. A Complete Definition of Data Flow Diagram Environment for Metaview. Department of Computing Science, University of Alberta, May 1994.
- [Fin93e] Piotr Findeisen. The Metaview Software. Department of Computing Science, University of Alberta, March 1993.
- [Fin93d] Piotr Findeisen. Project Daemon Reference Manual. Department of Computing Science, University of Alberta, March 1993.
- [Fin93c] Piotr Findeisen. Environment Definition for Database Engine. Department of Computing Science, University of Alberta, May 1993.
- [Fin93b] Piotr Findeisen. Environment Definition Guide or How to Build an Environment Description for Metaview. Department of Computing Science, University of Alberta, March 1993.
- [Fin93a] Piotr Findeisen. Aggregation in the EARA Model. Department of Computing Science, University of Alberta, April 1993.

- [Fin92b] Piotr Findelsen. The Graphical Extension for EARA Model. Department of Computing Science, University of Alberta, September 1992.
- [Fin92a] Piotr Findelsen. The EARA\GE Model for Metaview. Department of Computing Science, University of Alberta, November 1992.
- [DeD91] J. M. DeDoncker, P.G. Sorenson, and J.P. Tremblay. Metasystems for Information Processing System Specification Environments. *Information Systems (3)*:331-337, August 1991.
- [GCT92] Dennis de Champeaux and Penelope Famer. A Comparative Study of Object-Oriented Analysis Methods. *Journal of Object-Oriented Programming*, pages 21-33, 1992.
- [CY91] Peter Code and Edward Yourdon. *Object-Oriented Analysis*. 2nd ed. Yourdon Press, 1991.
- [Gol93] Derek Coleman, Patrick Arnold, Stephanie Bodoff, and et al. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1993.
- [CN89] M. Chen and J. F. Nunnemaker. MetaPlex: an integrated environment for organization and information System Development. *Proceedings of Tenth International Conference on Information Systems*, pages 141-151, 1989.
- [Bud91] Timothy Budd. An Introduction to Object-Oriented Programming. Addison Wesley, 1991.
- [Boo94] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1994.
- [Bol92] Germain Bollox, Paul G. Sorenson, and J. Paul Tremblay. Transformations using a meta-system approach to software development. *Software Engineering Journal*, pages 425-437, November 1992.

## Bibliography

- **Supporting Integrated Methodology and Integrated Environments**  
Previous work has been done to support multiple views for an integrated specification environment [Zhu93] and to support transformation processes between different structured methods [Lee92]. It would be challenging and valuable work to explore multiple-view environments that support process transformations between different O-O methods, and furthermore, between the Structured Methods and Object-Oriented Methods for method conversion.

features. Metaview's facilities to support across-model constraints were investigated for the first time in this thesis. We feel these can also be used in other non-O-O method environments.

The OOM Method, an enhanced O-O method, was proposed and prototyped in Metaview. This method combines several good features of different O-O methods in a manner that allows the analyst/designer to choose a set of models according to the size of the application system.

### 2. Critiquing the Representative O-O Methods

The representative O-O methods: OMT, Fusion Method, Booch's Method, and Shlaer-Mellor's Method were critically reviewed and their shortcomings were identified. This was helpful in choosing good O-O methods and/or developing better O-O methods. Because of such a comparison and critique, the future modeling of these O-O methods will be made easier.

### 3. Improvement of Metaview's Capability

Based on the modeling experience in this thesis, the ECL has been enhanced to provide new features such as using lexical conventions [Fin94b] to check an arbitrary string [Fin94c] which should be enclosed in a pair of parentheses.

## 6.2 Suggestions of Future Research

There are several areas of future research and prototype development:

- Evolving the OOM Method

The OOM Method incorporate many of the advantages of different O-O methods. However, this method has not been used to develop anything but small scale application system specifications. New features will be undoubtedly added to the OOM Method in the future. In particular, how to transform the OOD models to the implementation code needs to be explored much beyond the brief guideline given in chapter 5.

- Improving Metaview's Facilities

As mentioned in section 6.1, Metaview's modeling limitations, such as the EDL\ECL's expression capability and MGED's editing function, should be improved.

## ● MGED's Functions

The Metaview Graphical Editor (MGED) provides the basis for a user-friendly graphical interface; however, the following functions should be provided:

- Grouping and Scaling Entities  
The ability to group components of a diagram for the purposes of moving, copying, and deleting is required. In addition, an entity icon should be scalable.
- Encapsulating Entities  
During the O-O modeling, it may be desirable to encapsulate several existing entities and their relationships, which are part of a diagram, to a representative abstract entity (called an aggregate entity in the EAVRA model). This “bottom-up” or “synthesis” approach to aggregation is not currently supported.

## ● Dynamically Defining Properties of Graphical Objects

The entity icons are defined in a graphical environment using EDL. If it is desirable to change the shape of an icon, the environment definition must modify the environment definition code, compile it and generate the environment again. The ability to change dynamically certain properties of graphical objects (e.g., size, color) should be supported.

## Contributions

The major contributions of the thesis are as follows:

### 1. Fulfilling the Thesis Objectives

After investigating the feasibility of prototyping O-O methods using Metaview, it was found that Metaview was powerful enough to generate most of essential features of O-O specification environments. Particularly, the object, class, inheritance, and aggregation etc are generated. The strengths and limitations of Metaview's modeling capability were studied and based on this assessment, several improvements to Metaview were proposed. Some features, such as multiple aggregation, have not been provided in Metaview. The thesis addresses how to model O-O methods without the support of these

Metaview supports various constraints for specification environments. The completeness and consistency checks in each O-O model are implemented in both the TMO and OOM environments. Support for user-initiated transitions between different models, especially from OOM models to OOD models is provided by defining constraints across different models in the OOM support environment. This across-model constraint capability is not commonly found in current CASE tools such as Cadre's Teamwork [Inc93a, Inc93b].

- Metaview Provides a Common Transparent Repository  
Metaview provides a scheme to store all the analysis and design results produced in support environments that are prototyped using Metaview. The repository is transparent to the analyst/developer.
- Metaview Supports the Study of Methods  
The OOM Method is the result of studying different O-O methods. The OOM's prototype in Metaview is a very useful initial step in a continuous process of improvement for the OOM Method.

#### Some Metaview's limitations are:

- EDL/ECL Support  
Although the new version of EDL/ECL is available [Gaq94b, Fin94b], full-compiler support for EDL/ECL is still needed. EDL should be extended to support multiple aggregation. At present, an entity can only be exploded (or aggregated) to one aggregate. This makes it difficult to model a complex O-O Method, to strongly connect a subsystem with its static aspects, a corresponding Object Analysis Model should be formed through aggregation for the subsystem. To strongly connect the same subsystem with its dynamic aspects, another aggregate, called State Model, should be constructed. Currently, a subsystem can only be exploded to one aggregate. The second connection, say the one between a subsystem and its State Model, has to be modeled in an indirect way. Another entity called `state_model_name` is created to connect the entity subsystem to its aggregate State Model via a new created relationship called `subsystem_explode`.

existing environment definitions (conceptual, graphical and constraint-based).  
 Section 6.1 summarizes the thesis conclusions and contributions regarding to the  
 two objectives. Section 6.2 discusses future research.

## 6.1 Conclusions and Contributions

### Conclusions

Based on the experience of prototyping both OMT and our enhanced OOM  
 Method using Metaview, the advantages and disadvantages of this metamodel's mod-  
 eling capability are presented.

#### Metaview's major strengths:

- Metaview Provides Strong Facilities to Define O-O Environments
- Metaview's EARA model supports quite well two key functions, aggregation and  
 classification, in defining an O-O environment. A major difficulty in modeling  
 an O-O environment is the representation of nested diagrams. For example, in  
 OMT's State Model, a state may have a nested diagram to show its substates.  
 All the neighbor states and the input to/output from that state should be  
 inherited by its nested diagram. The aggregation function provides a simple  
 way to define nesting. An entity can be easily exploded into an aggregate  
 containing a number of entities and their relationships, with its neighbors and  
 input/output being copied into the aggregate. The semantics for aggregation  
 are well defined and consistently applied in a Metaview support environment.
- A large number of entities and their relationships may exist in an O-O method.  
 Many of them may share common characteristics. Classification makes O-O  
 modeling efficient because the subtyping relationships can be created between  
 those entities and relationships which share common characteristics.
- The OMT's definitions of entity and relationship types, and in particular the  
 constraints are heavily reused in OOM support environment. In addition, some  
 of the definitions of OOM's models are also reused in the other models. Reuse  
 in Metaview largely eases the work of defining a new O-O environment.
- Metaview Provides Strong Facilities to Define Various Constraints



## Chapter 6

### Conclusions and Future Research

Both O-O methods and meta CASE systems to support the development of software engineering environments are in a state of rapid evolution. This has presented a major challenge to our study of the object-oriented modeling in Metaview. The two primary goals that motivated this research are:

- To Investigate Metaview's Modeling Capability for Object-Oriented Methods
- Metaview was previously shown to be capable of modeling several traditional structured methods, such as Data Flow Diagrams, Structure Chart, and ADISSA, etc, but until now O-O specification environments were not created using Metaview. With the increased popularity of O-O methods, it is important to investigate how complete and consistent the O-O methods can be modeled using Metaview. Metaview's advantages and disadvantages observed from this modeling experience are essential for the further improvement and development of the Metaview prototype.
- To Propose An Enhanced O-O Method and Model it Using Metaview
- There are a number of different O-O development methods each with some shortcomings. We believe it not sufficient to investigate Metaview's capability by only prototyping various existing O-O methods. To provide a broader test of Metaview's capability, it was decided to prototype a representative method which included the most important advantages of current O-O methods. Therefore, the second objective of this thesis was to propose an enhanced O-O method and implement it in our metasystem while trying to reuse as much as possible

constant clause. A constraint can be shared by various O-O models if their names are defined as the parameters of the same constraint.

Most OMT environment definitions are reused and modified for the new support environment. Based on the experience of modeling OMT method, the new entity types, relationship types and constraints are easier to build because of the increased Metaview expertise of the model builder. However, two major limitations of Metaview are observed. The transformations between different models are manual because the current Metaview prototype does not support copying and other forms of automatic transformations. To ease the analyst's work, the automatic or semi-automatic transformations should be provided. The transformations between different software specification environments have been studied [Bolos, 1992], which are helpful in constructing the automatic transformations between different models within an O-O support environment. Another major limitation is that Metaview's universal graphical editor MGED does not provide a query capability or some better navigation methods to locate a diagram. This would be particularly helpful in the analysis and design of a large, complex system, because lots of software specification diagrams are created.

Measurement	Models			Entity Types #			Relationship Types #			Constraints #			Coding Hours
	new	modified	removed	new	modified	removed	new	modified	removed	new	modified	removed	
Process	6			3			9			9			12
State	8			2			8			8			20
Object Analysis	12			32			32			32			60

Table 2.2: The measurement of OMT modeling

Measurement	Models			Entity Types #			Relationship Types #			Constraints #			Coding Hours
	new	modified	removed	new	modified	removed	new	modified	removed	new	modified	removed	
Subsystem Design	1			11			16			28			8
Object Design	1			12			18			28			8
Subsystem Visibility	2			3			3			4			2
Object Visibility	6			3						2			8
Subsystem Communication				2			1			3			2
Object Communication	3			2			4			10			10
Process		1		2			3			9			2
State		1		7			2			8			2
Object Analysis		3		9			22			18			8
Subsystem Analysis	2			8			12			28			8
Domain Analysis	1			4			3			18			2

Across-model Constraints # = 10, Coding Hours = 6

Table 2.3: The measurement of OOM modeling

how to transform from the OOD models to OOP have not been considered in detail.

3. The OOM support environment has been successfully defined using Metaview. However, only small scale application systems have been analysed and designed using the OOM Method.

As a result of defining the OOM Method, it is observed that Metaview has strong modeling capabilities for defining O-O methods. A number of entity and relationship types have been defined in OMT and OOM support environments. Moreover, Metaview's constant definition capability as defined in the new ECL [Fin94] is powerful enough to support the completeness and consistency checks within models and across different models.

The pre-defined OMT models are heavily reused in the OOM support environment. As illustrated in table 2.2 and table 2.3, all the entity types, relationship types and constants originally defined in OMT's Object Model, Dynamic Model and Functional Model are reused or modified in modeling OOM's Object Analysis Model, State Model and Process Model, respectively. The workload of defining OOM support environment is largely decreased because of the reuse in Metaview. The reuse also exists among different OOM models. Many definitions in subsystem-level models are reused in the system-level models. For example, eight out of ten entity types, fifteen out of sixteen relationship types, and all the (twenty-eight) constants in the Subsystem Analysis Model are reused from the Object Analysis Model. The transformation relations between different models as shown in Figure 34 help to decide how to reuse definitions from other models. For example, the Object Design Model is transformed from the Object Analysis Model, Object Communication Model, Object Visibility Model, and the definitions in the Object Design Model are heavily reused from those four models: twelve out of thirteen entity types, all the (eighteen) relationship types and (twenty-eight) constants are reused.

Comparing the coding hours for defining OMT and OOM support environments, it is observed that the workload of OOM coding is not heavy. The major reason is Metaview's support for the reuse. First, to define an O-O model, an aggregate is used to include all the entity and relationship types defined in this O-O model. The pre-defined entity and relationship types are easy to redefine and include in the model aggregate. Second, O-O model's name is used as an input parameter for each

constraints within models and across models.

Because Metaview is an evolving prototype, the environment designer may encounter problems in defining new features for an O-O environment. For example, OOM required the across-model constraints and such constraints were not modeled in the existing environments. The identification of difficulties in defining new features using Metaview provides important feedback in the evaluation and evolution of Metaview's EARA meta-model. For example, based on the thesis work, support for multiple aggregates in the EARA model is now being considered.

In summary, defining an O-O environment can be divided into three phases:

- Design the O-O method by defining various O-O models which include the entity and relationship types and constraints for each model. It is important in this phase to model the O-O method faithfully.
- Optimize the models for an O-O method by using subtyping and aggregation and reusing parts of other previously defined environments. As an example of this, during the modeling of OOM method, significant effort went into the adoption and reuse of OMT model definitions.
- Iteratively test and refine the environment during implementation.

## 2.6 Summary

The goals of the OOM Method are to make software development a rational, controlled, and systematic process. The OOM Method is summarized as follows:

1. This method is an enhancement of current popular O-O methods based on desirable aspects from the OMT, Fusion, Booch, and Shlaer-Mellor's Method. It is proposed with two major objectives: to provide a flexible method to develop various systems which range in size and complexity, and to support a well-defined transformation between OOA and OOD models. Completeness and consistency checking are provided not only within a model, but across different models in support of transformations.

2. To develop a software system, the work is divided into three steps: OOA, OOD and OOP. The OOM Method supports the first two phases. A brief guide of

```

PARTICIPANTS (process, data_object, process|actor|data_store)
ROLE(source, flow, destination)
data_flow GENERIC IS_A universal_relationship

% Relationship types defined in the Process Model
RELATIONSHIP_TYPE

data_object GENERIC IS_A universal_entity;
actor GENERIC IS_A universal_entity;
data_store GENERIC IS_A universal_entity;

BECOMES process_model;
semantics: text, postconditions: text;
ATTRIBUTES (arguments: text, preconditions: text,
process GENERIC IS_A universal_entity;

% Entity types defined in the Process Model
ENTITY_TYPE

ATTRIBUTES (description: text, system_level: boolean);
state_action, send_event)
data_flow, fcontrol_flow, oneValue_store)
%relationship
COMPONENTS (process, data_object, data_store, actor,
AGGREGATE_TYPE process_model

/*-----*/
Process Model
/*-----*/

message("The final state", entity.name,
OTHERWISE
SATISFY rel->destination == entity
ALL rel FROM [control_flow @ aggregate]
ALL entity FROM [final_state @ aggregate]
CONSTRAINT (8) final_state_participating
(aggregate: state_model)
"should not be isolated.");

```

should be included by a pair of brackets.");

```
CONSTRAINT (8) transitionName_participating
(aggregate: state_model)
ALL entity FROM [transition_name @ aggregate]
ALL rel FROM [control_flow @ aggregate]
SATISFY rel->flow == entity
OTHERWISE
message("The transition name", entity.name,
"should not be isolated.");
```

```
CONSTRAINT (8) relName_participating
(aggregate: state_model)
ALL entity FROM [rel_name @ aggregate]
ALL rel FROM [send_event @ aggregate]
SATISFY rel->relation == entity
OTHERWISE
message("The Relation", entity.name,
"should not be isolated.");
```

```
CONSTRAINT (8) activity_participating
(aggregate: state_model)
ALL entity FROM [activity @ aggregate]
ALL rel FROM [state_activity @ aggregate]
SATISFY rel->inactivity == entity
OTHERWISE
message("The activity", entity.name, "should not be isolated.");
```

```
CONSTRAINT (8) action_participating
(aggregate: state_model)
ALL entity FROM [action @ aggregate]
ALL rel FROM [state_action @ aggregate]
SATISFY rel->inaction == entity
OTHERWISE
message("The action ", entity.name, "should not be isolated.");
```

```
CONSTRAINT (8) initialState_participating
(aggregate: state_model)
ALL entity FROM [initial_state @ aggregate]
ALL rel FROM [control_flow @ aggregate]
SATISFY rel->source == entity
OTHERWISE
message("The initial state", entity.name,
```

```

message("The transition name", entity.name, "'s event_condition
SATISFY properly enclosed ('[', ']', condition)
WHERE condition = entity.event_condition
ALL entity FROM [transition_name @ aggregate
aggregate: state_model)
CONSTRAINT (4) transitionName_checking2

should be input as the form of \action.";
message("The transition name", entity.name, "'s action
OTHERWISE
SATISFY check_action(action)
WHERE action = entity.action_name
ALL entity FROM [transition_name @ aggregate
aggregate: state_model)
CONSTRAINT (4) transitionName_checking1

firstChar == '\\';
SATISFY
firstChar = _head(list)
WITH list = _elements(str);
PREDICATE check_action(VALUE str: string)
% Constraints defined in the State Model

PARTICIPANTS (intermediate_state, action);
ROLES(state, inAction)
state_action GENERIC IS_A universal_relationship

PARTICIPANTS (intermediate_state, activity);
ROLES(state, inActivity)
state_activity GENERIC IS_A universal_relationship

PARTICIPANTS (transition_name, rel_name, class);
% rel_name and class have been modeled in the Object Diagram
ROLES(source, flow, destination)
send_event GENERIC IS_A universal_relationship

intermediate_state|final_state);
PARTICIPANTS
ROLES(source, flow, destination)
control_flow GENERIC IS_A universal_relationship

```



```

% Relationship types defined in the State Model
RELATIONSHIP_TYPE
% Relationship types defined in the State Model
    action GENERIC IS_A universal_entity;
    activity GENERIC IS_A universal_entity;
    event_attribute: text;
    ATTRIBUTES (action_name: string, event_condition: string,
        transition_name GENERIC IS_A universal_entity
        final_state GENERIC IS_A state;
        BECOMES state_model;
        intermediate_state GENERIC IS_A state;
        initial_state GENERIC IS_A state;
        state GENERIC IS_A universal_entity;
ENTITY_TYPE
% Entity types defined in the State Model
    ATTRIBUTES (description: text, system_level: boolean);
    state_action, send_event)
    control_flow, state_activity,
    %relationship types
    COMPONENTS (state, transition_name, activity, action,
        AGGREGATE_TYPE state_model
%-----*
State Model
%-----*

message("The", entity.name, "should connect to a class/object.");
OTHERWISE
SATISFY rel->class == entity
    ALL rel FROM [derived_relation @ aggregate]
    [derived_object @ aggregate]
    + [derived_class @ aggregate]
    (aggregate: object_analysis_model)

```

```

message("The partOf_center does not have sub-entity.");

CONSTRAINT (8) isaCenter_participating
(aggregate: object_analys_model|object_design_model|
subsystem_analys_model|subsystem_design_model)
ALL entity FROM [isa_center @ aggregate]
ALL rel FROM [generalize_from @ aggregate]
SATISFY rel->isa_structure == entity
OTHERWISE
message("The isa_center does not have super-entity.");

CONSTRAINT (8) isaCenter_participating
(aggregate: object_analys_model|object_design_model|
subsystem_analys_model|subsystem_design_model)
ALL entity FROM [isa_center @ aggregate]
ALL rel FROM [generalize_to @ aggregate]
SATISFY rel->isa_structure == entity
OTHERWISE
message("The isa_center does not have sub-entity.");

CONSTRAINT (8) dualifier_participating
(aggregate: object_analys_model|object_design_model|
subsystem_analys_model|subsystem_design_model)
ALL entity FROM [dualifier @ aggregate]
ALL rel FROM [dualifier_relate @ aggregate] +
[dualified_associate @ aggregate]
SATISFY rel->attaching == entity;
OTHERWISE
message("The Dualifier", entity.name,
"should not be isolated.");

CONSTRAINT (8) comment_participating
(aggregate: object_analys_model|object_design_model|
domain_analys_model|subsystem_analys_model|
subsystem_design_model)
ALL entity FROM [comment @ aggregate]
ALL rel FROM [comment_relate @ aggregate]
SATISFY rel->attaching == entity
OTHERWISE
message("The comment", entity.name,
"should not be isolated.");

CONSTRAINT (8) derived_class_participating

```

```

    subquery design_model
    ALL entity FROM [oo_operation @ aggregate]
    ALL rel FROM [entity_operation @ aggregate]
    SATISFY rel->operation_name == entity
    OTHERWISE
    message("The operation", entity.name,
    "should belong to an entity");

CONSTRAINT (8) ternary_center_participating
    (aggregate: object_analysis_model|object_design_model|
    subsystem_analysis_model|subsystem_design_model)
    ALL entity FROM [ternary_center @ aggregate]
    ALL rel FROM [ternary_associate @ aggregate]
    SATISFY rel->ternary_structure == entity
    OTHERWISE
    message("The ternary_center should not be isolated.");

CONSTRAINT (8) link_attr_participating
    (aggregate: object_analysis_model|object_design_model|
    subsystem_analysis_model|subsystem_design_model)
    ALL entity FROM [rel_attribute @ aggregate]
    ALL rel FROM [link_attr_relate @ aggregate] +
    [link_attr_ternary @ aggregate]
    SATISFY rel->attaching == entity
    OTHERWISE
    message("The link attribute should not be isolated.");

CONSTRAINT (8) part_of_center_participating
    (aggregate: object_analysis_model|object_design_model|
    subsystem_analysis_model|subsystem_design_model)
    ALL entity FROM [part_of_center @ aggregate]
    ALL rel FROM [aggregate_from @ aggregate]
    SATISFY rel->part_of_structure == entity
    OTHERWISE
    message("The part_of_center does not have super-entity.");

CONSTRAINT (8) part_of_center_participating
    (aggregate: object_analysis_model|object_design_model|
    subsystem_analysis_model|subsystem_design_model)
    ALL entity FROM [part_of_center @ aggregate]
    ALL rel FROM [aggregate_to @ aggregate]
    SATISFY rel->part_of_structure == entity
    OTHERWISE

```

```

    CONSTRAINT (4) association_before_propagation
    (aggregate: object_analysia_model|object_design_model|
    subsystem_analysia_model|subsystem_design_model)
    ALL rel FROM [propagating_operation @ aggregate]
    SATISFY entity_has_association(rel->sender, rel->receiver);
    entity_has_association(rel->receiver, rel->sender)
    OTHERWISE
    message(rel->receiver.name, "and", rel->sender.name,
    "should have association before the operation_propagation");

    CONSTRAINT (8) object_participating
    (aggregate: object_analysia_model|object_design_model)
    ALL entity FROM [object @ aggregate]
    ALL rel FROM [link @ aggregate]
    SATISFY rel.object == entity
    OTHERWISE ALL entity FROM [object @ aggregate]
    ALL rel FROM [instant @ aggregate]
    SATISFY rel.object == entity
    OTHERWISE
    message(entity.name, "is not linked to any class or object.");

    CONSTRAINT (8) attribute_participating
    (aggregate: object_analysia_model|object_design_model|
    domain_analysia_model|subsystem_analysia_model|
    subsystem_design_model)
    ALL entity FROM [oo_attribute @ aggregate]
    ALL rel FROM [entity_attribute @ aggregate]
    SATISFY rel->attribute_name == entity
    OTHERWISE
    message("The attribute", entity.name,
    "should belong to an entity");

    CONSTRAINT (8) operation_participating
    (aggregate: object_analysia_model|object_design_model|
    domain_analysia_model|subsystem_analysia_model|
    subsystem_design_model)
    ALL rel FROM [operation @ aggregate]
    SATISFY rel->operation_name == entity
    OTHERWISE
    message("The operation", entity.name,
    "should belong to an entity");

    CONSTRAINT (8) association_has_association
    (entity: any_level, entity2: any_level)
    ALL rel FROM [association @ *]
    SATISFY
    rel.class == entity1
    rel.class == entity2;

```

```

    aggregate: object_analys_model|object_design_model|
domain_analys_model|subsystem_analys_model|
    subsystem_design_model)
    ALL entity FROM [rel_name @ aggregate]
    ALL rel FROM [associate @ aggregate] + [link @ aggregate] +
    [qualified_associate @ aggregate] + [instant @ aggregate] +
    [derived_relation @ aggregate] + [domain_link @ aggregate]
    SATISFY rel->relation == str
    OTHERWISE
    message("Relation name", entity.name, "should not be isolated.");

CONSTRAINT (8) constraint_participating
    aggregate: object_analys_model|object_design_model|
    subsystem_analys_model|subsystem_design_model)
    ALL entity FROM [constraint @ aggregate]
    ALL rel FROM [general_constraint @ aggregate]
    SATISFY rel->constraint_name == str
    OTHERWISE
    message("Constraint name", entity.name, "should not be isolated.");

CONSTRAINT (8) operation_participating
    aggregate: object_analys_model|object_design_model|
    subsystem_analys_model|subsystem_design_model)
    ALL entity FROM [oo_operation @ aggregate]
    ALL rel FROM [propagating_operation @ aggregate] +
    [entity_operation @ aggregate]
    SATISFY rel->operation_name == str
    OTHERWISE
    message(entity.name, "should not be isolated.");

PREDICATE entity_has_operation
    (sender: all_level, operation: oo_operation)
    ALL rel FROM [entity_operation @ *]
    SATISFY
    rel.entity_body == sender
    rel.operation_name == operation;

CONSTRAINT (4) operation_from_sending_class
    aggregate: object_analys_model)
    ALL rel FROM [propagating_operation @ aggregate]
    SATISFY entity_has_operation(rel->sender, rel->operation_name)
    OTHERWISE
    message(rel->operation_name, "should not be isolated.");

```

```

CONSTRAINT (8) rename_participating

    "should be in the form of {...}.";
    message("The description of the constraint", entity.name,
    OTHERWISE
    SATISFY properly_enclosed('{', '}', content)
    WHERE content = entity.real_content
    ALL entity FROM [constraint @ aggregate]
    subsystem_analysais_model|subsystem_design_model
    aggregate: object_analysais_model|object_design_model|
    CONSTRAINT (4) constraint_entity_checking
    % Constraints defined in the Object Analysis Model, some are reusable

    aggregate_to_optional IS_A aggregate_to;
    aggregate_to_one IS_A aggregate_to;
    aggregate_to_many IS_A aggregate_to;

    PARTICIPANTS(partOf_center, class|subsystem|b_class);
    ROLES(partOf_structure, component)
    aggregate_to GENERIC IS_A universal_relationship

    optional_aggregate_from IS_A aggregate_from;
    one_aggregate_from IS_A aggregate_from;
    many_aggregate_from IS_A aggregate_from;

    PARTICIPANTS(class|subsystem|b_class, partOf_center);
    ROLES(class, partOf_structure)
    aggregate_from GENERIC IS_A universal_relationship

    % 3. aggregation
    ATTRIBUTES (role_name: string, role_name: string);
    PARTICIPANTS(isa_center, class|subsystem|b_class)
    ROLES(isa_structure, subclass)
    generalize_to GENERIC IS_A universal_relationship,

    ATTRIBUTES (role_name: string, role_name: string);
    PARTICIPANTS(class|subsystem|b_class, isa_center)
    ROLES(superclass, isa_structure)
    generalize_from GENERIC IS_A universal_relationship,

    % 2. inheritance
    qualified_many_to_one IS_A associate;

```

```

qualified_many_to_many IS_A associate;

ATTRIBUTES (role_name: string, role_name: string);
class|subclass|d_class)
PARTICIPANTS(qualifier, rel_name,
ROLES(attachng, relng, class)
qualifier_relate GENERIC IS_A universal_relationship
% 1. qualified association
% indirect relationship types
optional_to_optional_associate IS_A associate;
one_to_optional_associate IS_A associate;
one_to_one_associate IS_A associate;
many_to_optional_associate IS_A associate;
many_to_one_associate IS_A associate;
many_to_many_associate IS_A associate;

ATTRIBUTES (role_name: string, role_name: string);
(d_subsystem, rel_name, d_subsystem)
(d_class, rel_name, d_class)
PARTICIPANTS(class, rel_name, class)
ROLES(class, relation, class2)
associate GENERIC IS_A universal_relationship
% complex relationship types
(rel_name, constraint, rel_name);
PARTICIPANTS(class, constraint, class)
ROLES(entity_from, constraint_name, entity_to)
general_constraint GENERIC IS_A universal_relationship

PARTICIPANTS(rel_attribute, ternary_center);
ROLES(attachng, attached)
linker_ternary GENERIC IS_A universal_relationship
PARTICIPANTS(rel_attribute, rel_name);
ROLES(attachng, attached)

```

```

linkAttr_relate GENERIC IS_A universal_relationship

subsystem|b_subsystem|b_class|rel_name);
isA_center|partOf_center|ternary_center|
PARTICIPANTS(comment, class|object|derived_class|derived_object|
ROLES(attachng, attached)
comment_relate GENERIC IS_A universal_relationship

PARTICIPANTS(class, oo_operation, class);
ROLES(sender, operation_name, recipient)
propagating_operation GENERIC IS_A universal_relationship

PARTICIPANTS(object, rel_name, class);
ROLES(instance, relation, class)
instance generic is_a universal_relationship

PARTICIPANTS(class, ternary_center);
ROLES(class|subsystem|b_subsystem|b_class, ternary_structure)
ternary_association GENERIC IS_A universal_relationship

PARTICIPANTS(object, rel_name, object);
ROLES(object, relation, object2)
link GENERIC IS_A universal_relationship

(object, rel_name, derived_object);
PARTICIPANTS(class, rel_name, derived_class)
ROLES(class, relation, class2)
derived_relaton GENERIC IS_A universal_relationship

domain|subsystem|b_class|b_subsystem, oo_operation);
(class|derived_class|object|derived_object|rel_operation|
PARTICIPANTS
ROLES(entity_body, operation_name)
entity_operation GENERIC IS_A universal_relationship

domain|subsystem|b_class|b_subsystem, oo_attribute);
(class|derived_class|object|derived_object|rel_attribute|
PARTICIPANTS
ROLES(entity_body, attribute_name)
entity_attribute GENERIC IS_A universal_relationship

% simple relationship types
RELATIONSHIP_TYPE

```



```

% Relationship types defined in the Object Analysis Model

rel_name GENERIC IS_A universal_entity;
ATTRIBUTES (rel_content: string);
constraint GENERIC IS_A universal_entity

comment GENERIC IS_A universal_entity;

rel_attribute GENERIC IS_A universal_entity;

partOf_center GENERIC IS_A control_center;
isa_center GENERIC IS_A control_center;

ternary_center GENERIC IS_A control_center;

control_center GENERIC IS_A universal_entity;

qualifier GENERIC IS_A universal_entity;

oo_attribute GENERIC IS_A universal_entity;

semantics: text, postconditions: text);
export_control: (public, protected, private, implementation),
ATTRIBUTES (arguments: text, preconditions: text,
oo_operation GENERIC IS_A universal_entity

derived_object GENERIC IS_A object;

ATTRIBUTES (responsibilities: string, constraints: string);
derived_class GENERIC IS_A data_universal

object GENERIC IS_A universal_entity
ATTRIBUTES (responsibilities: text, constraints: text);

BECOMES state_analysis_model;
ATTRIBUTES (responsibilities: string, constraints: string)

class GENERIC IS_A universal_entity

```

% Entity types defined in the Object Analysis Model  
ENTITY\_TYPE

# A.1 Conceptual Definition for Three OOA Models

```

ENVIRONMENT_NAME oom;

/*-----
Generic Types and Predicates
-----*/

ENTITY_TYPE universal_entity GENERIC
ATTRIBUTES (description: text);

RELATIONSHIP_TYPE universal_relationship GENERIC
ATTRIBUTES (description: text);

PREDICATE last_char_is (VALUE char, VALUE SET str)
IF length(str) = 1 THEN head(str) = c
ELSE last_char_is (VALUE char, tail(str));

PREDICATE properly_enclosed
(VALUE char1, VALUE char2, VALUE str: string)
WITH list = elements(str);
char = head(list)
SATISFY
char = char1 AND last_char_is(char2, list);

/*-----
Object Analysis Model
-----*/

AGGREGATE_TYPE object_analysis_model
COMMENTS (class, object, derived_class, derived_object,
oo_attribute, oo_operation, qualifier, control_center,
rel_attribute, comment, constraint, rel_name,
% relationship types
entity_attribute, entity_operation, derived_relation,
link, ternary_associate, instance, comment_relate,
propagating_operation, associate, linkAttr_relate,
linkAttr_ternary, general_constraint, qualifier_relate,
qualified_association, generalize_from, generalize_to,
aggregate_from, aggregate_to)
ATTRIBUTES (description: text);

```

## Appendix A

### Environment Part of the Definition for OOM Support

The OOM Method is rather complex and the definition is very long. Therefore it was decided to include only a representative part of the definition in the appendix.

Because they form a basis of other OOM models, the Object Analysis Model, State Model, and Process Model are important models in both TMO and OOM. A complete concept-level definition for these three models is presented first in section A1.

The OOM's graphical aspect is defined using EDL/ECL's graphical extension (GE). As mentioned in this thesis, there are some limitations in defining OOM support environment's graphical aspect. Only the Process Model's graphical environment definition is presented in section A2.

Metaview's across-model constraints are first studied in the OOM support environment. They are defined on two levels: the OOM's subsystem and system levels. Because system-level constraints are quite similar to the subsystem-level ones (refer to chapter 5), only the subsystem-level across-model constraints are presented in subsection A3 of the appendix.

OW

20-10



```

OTHERWISE
message("The class", class.name, "in object visibility
model", agr2.name, "should have been captured
by the corresponding analysis model.");
class.name == class.name
SOME class FROM [class@agr1] SATISFY
ALL class FROM [v_class@agr2] SATISFY
*.name == agr2.analysis_model_name]
ALL agr1 FROM [object_analysis_model:
(agr2: object_visibility_model)
CONSTRAINT (e) visibility_model_complete
Must be included in the Object Analysis Model.
All the classes depicted in the Object Visibility Model
object_design_model.");
agr1.name, "is not correctly duplicated to the
message("Server ", s_name.name, "in object_visibility_model",
OTHERWISE
_child(name2) == child1
SOME s_names FROM [server_name@agr2] SATISFY
WHERE child1 = _child(s_name1) SATISFY
ALL s_name1 FROM [server_name@agr1]
*.name == agr2.visibility_model_name]
ALL agr1 FROM [object_visibility_model:
CONSTRAINT (e) visibility_model_complete
Must be included in the Object Analysis Model.
All the classes depicted in the Object Visibility Model
% The entity type server_name may have exclusive servers depicted
% in its aggregate exclusive_servers. All the exclusive_servers
% depicted in the Object Visibility Model should be included in
% the Object Design Model.
```

### A.3 Some Across-Model Constraints

```

-----\
Subsystem-Level Across-Model Constraints
-----\

% All the class entities in the Object Analysis Model should be
% included in the Object Design Model.
CONSTRAINT (6) design_model_completed (args: object_design_model)
ALL args FROM [object_analysis_model:
*.name == args.analysis_model_name]
ALL class FROM [class@args] SATISFY
SOME class FROM [b_class@args] SATISFY
class.name == class.name
OTHERWISE
message("The class", class.name, "in object_analysis_model",
args.name, "should also be captured by current design model.");

% All the class entities depicted in the Object Visibility Model
% should be included in the Object Design Model.
CONSTRAINT (6) design_model_completed (args: object_design_model)
ALL args FROM [object_visibility_model:
*.name == args.visibility_model_name]
ALL class FROM [class@args] SATISFY
SOME class FROM [b_class@args] SATISFY
class.name == class.name
OTHERWISE
message("The class", class.name, "in object_visibility_model",
args.name, "should also be captured by current design model.");

% All the server_name entities depicted in the Object Visibility
% Model should be included in the Object Design Model.
CONSTRAINT (6) design_model_completed (args: object_design_model)
ALL args FROM [object_visibility_model:
*.name == args.visibility_model_name]
ALL s_name FROM [server_name@args] SATISFY
SOME s_name FROM [b_class@args] SATISFY
s_name.name == s_name.name
OTHERWISE
message("The server_name", s_name.name, "in object_visibility

```

```

CONSTRAINT (4) data_close_to_edge (GRAPHICAL diag: process_model)
ALL dataflow FROM [data_flow @ diag] + [control_flow @ diag] +
[oneValue_store @ diag]
WHERE d = dataflow->data,
xpos = _origin(d) + _width(d)\2,
ypos = _top(d) + _height(d)\2
SATISFY _edge_distance(dataflow, xpos, ypos) <= 50
OTHERWISE
message("The data must be located near the corresponding edge.");

CONSTRAINT (4) one_actor_one_icon (GRAPHICAL diag: process_model)
ALL icon FROM [actor @ diag]
ALL icon1 FROM [actor @ diag]
SATISFY icon == icon1
OTHERWISE
message("Each actor entity may be represented by only one icon.");

```



```

message("Each data flow icon can be used only in one edge.");
OTHERWISE
SATISFY dataflow1 == dataflow2
ALL dataflow2 FROM [data_flow @ diag: *->data == dataflow1->data]
ALL dataflow1 FROM [data_flow @ diag]
CONSTRAINT (4) no_duplicate_flow (GRAPHICAL diag: process_model)
% Constraints in the Process Model

PICTURES (hollowarrowhead AT destination));
LINKS (FROM source TO destination
destination AT (200,0))
flow AT (100, 20) PROPERTIES (,
NODES (source AT (0,0),
PROPERTIES (no_stretch = y)
EDGE_TYPE data_flow

PICTURES (arrowhead AT destination));
LINKS (FROM source TO destination
destination AT (200,0))
flow AT (100, 20) PROPERTIES (,
NODES (source AT (0,0),
PROPERTIES (no_stretch = y)
EDGE_TYPE fcontrol_flow

PICTURES (arrowhead AT destination));
LINKS (FROM source TO destination
destination AT (200,0))
flow AT (100, 20) PROPERTIES (,
NODES (source AT (0,0),
PROPERTIES (no_stretch = y)
EDGE_TYPE data_flow

LABELS (name AT (40, 20) PROPERTIES (x_size = 70, y_size = 30));
PICTURES (actor_pic AT (0, 0) ROTATED 0)
PROPERTIES (x_size = 81, y_size = 41)
ICON_TYPE actor_icon

LABELS (name AT (40, 14) PROPERTIES (x_size = 70, y_size = 20));
PICTURES (data_store_pic AT (0, 0) ROTATED 0)
PROPERTIES (x_size = 81, y_size = 41)

```

## A.2 Process Model's Graphical Definition

```

-----\
Process Model: Graphical Environment Definition
-----*/

CONSTANT DiagramWidth = 600; % Diagram Width
CONSTANT DiagramHeight = 770; % Diagram Height

DIAGRAM_TYPE process_model
PROPERTIES (x_size = DiagramWidth, y_size = DiagramHeight);

PICTURE_TYPE process_pic
ARC CENTER (40,20) RADIUS (40,20) START 0 SPAN 360;

PICTURE_TYPE data_store_pic
LINE FROM (0, 0) TO (80, 0)
LINE FROM (0, 40) TO (80, 40);

PICTURE_TYPE actor_pic
RECTANGLE FROM_CORNER (0,0) TO_CORNER (40, 80);

CONSTANT arrow_width 15;
CONSTANT arrow_height 5;

PICTURE_TYPE arrowhead
LINE FROM (-arrow_width, -arrow_height) TO (0, 0)
LINE FROM (-arrow_width, arrow_height) TO (0, 0);

PICTURE_TYPE hollowArrowhead
LINE FROM (-arrow_width, -arrow_height) TO (0, 0)
LINE FROM (-arrow_width, arrow_height) TO (0, 0)
LINE FROM (-arrow_width, -arrow_height)
TO (-arrow_width, arrow_height);

ICON_TYPE process_icon
PROPERTIES (x_size = 80+1, y_size = 40+1)
PICTURES (process_pic ROTATED 0 AT (0, 0) PROPERTIES ())
LABELS ( name AT (40, 14)
PROPERTIES (x_size = 55, y_size = 26));

ICON_TYPE data_store_icon

```

```

message("Actor", entity.name, "should not be isolated.");
OTHERWISE
rel->destination == entity
SATISFY rel->source == entity;
All rel FROM [data_flow @ aggregate]
ALL entity FROM [actor @ aggregate]
(aggregate: process_model)

```

```

CONSTRAINT (8) actor_participating
message("DataStore", entity.name, "should not be isolated.");
OTHERWISE
rel->destination == entity
SATISFY rel->source == entity;
[oneValue_store @ aggregate]
ALL rel FROM [data_flow @ aggregate] +
ALL rel FROM [data_store @ aggregate]
(aggregate: process_model)
CONSTRAINT (8) dataObject_participating
message("Process", entity.name, "should not be isolated.");
OTHERWISE
SATISFY rel->source == entity
[oneValue_store @ aggregate]
ALL rel FROM [data_flow @ aggregate] +
SATISFY rel->destination == entity
[control_flow @ aggregate]
(aggregate: process_model)
CONSTRAINT (8) process_participating
message("A process can not send to itself.");
OTHERWISE
SATISFY rel->source != rel->destination
ALL rel FROM [control_flow @ aggregate]
(aggregate: process_model)
CONSTRAINT (4) not_send_to_process_itself
% Constraints defined in the Process Model
PARTICIPANTS (process, data_object, data_store);
ROLE(source, flow, destination)
oneValue_store GENERIC IS_A universal_relationship
PARTICIPANTS (process, data_object, data_store);
control_flow GENERIC IS_A universal_relationship
PARTICIPANTS (process, data_object, process);
ROLE(source, flow, destination)
actor|data_store, data_object, process);

```